SECURITY AUDIT OF

ROCK ONYX SMART CONTRACTS

Public Report

Jun 06, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jun 06, 2024. We would like to thank the Harmonix Finance for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Rock Onyx Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Rock Onyx Smart Contracts

Rock Onyx is a platform which provides automated Vaults with various strategies to maximize your returns with low risk.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Rock Onyx Smart Contracts. It was conducted on commit `b7e50eb2aa37d1d9f8127a29dbc18a41ccdda1ce` from git repository *https://github.com/harmonixfi/rock-onyx-smart-contract*

The latest version of the following files were made available in the course of the review:

SHA-1 Sum	File
a59c6de164e273b0eada930024ece356472c65f2c3075aed562cbe204e500389	extensions/Aevo/Aevo.sol
c07b933fbdcbfa9bff5486258dcbb7d29971fe1370a496e74a1bd0f2ec2928be	extensions/Camelot/CamelotLiquidity.sol
1759436d3dc27e518af9c6c8aeab15850113a44774cd3ba1535b083502721369	extensions/Camelot/CamelotSwap.sol
a8c624776bacb281fda919fbde4f24c8cdb187904fcaf233941b92715512af56	extensions/Uniswap/Uniswap.sol
8fd7344c111e5d72d99cd83b24ef5599f2d2bc531199965e1700d31bf054cf96	extensions/Chainlink/PriceConsumer.sol
36483b418df4aecbfa783f72ffe1020b278ba9893253e98001e15d7884157c20	extensions/RockOnyxAccessControl.sol
6a76842fe2d953c49ff6e83629bcdbe526e779661b001949e178cec421c0bd67	extensions/TransferHelper.sol
eb164604af48afabb62847fc5cd9be26b90919b49808bccf50f6ae0988f128d2	lib/BaseSwap.sol
8b5f588f0c3f149ed989a15a44e96d9dbd6c0021358d76b9705890ab525fdce6	lib/FullMath.sol
f414bc9f1916d6bdd8f955a8b3fe30013ef0d137523b0cb004defb204110903e	lib/LiquidityAmounts.sol
332cb9b903821b48f7409d48cd727d83e5c9935d74a6f171e3305c8b2bdbb5dd	lib/ShareMath.sol

4a0887898a9346993cd9cbcc56a57225197ee995910e62de8970cbd80614fddb	vaults/stableUsdc/strategies/RockOnyxEthLiquidityStrategy.sol
5531755268fd9b1fc777839346355320491634a9d1564e10b3ce1c6504bed26e	vaults/stableUsdc/strategies/RockOnyxOptionsStrategy.sol
f594486c23214604a4b2410b1775dccf9dbed1cb6d1702e92c2d8cf088b7507b	vaults/stableUsdc/strategies/RockOynxUsdLiquidityStrategy.sol
40ef327a42e9ebe08e903d73fe5796d8c5f76fdea4c047c9ee12be9fe2eb2da8	vaults/stableUsdc/BaseRockOnyxOptionWheelVault.sol
317965c21f393f33d8e178b8386f453c7c913801de972588263a743c0dad6a66	vaults/stableUsdc/RockOnyxUSDTVault.sol

The Harmonix Finance team has been updated with the findings and recommendations. The team has acknowledged the issues and provided updates on the issues. The team has also responded to the findings and recommendations. The latest version of the code was reviewed on commit 20f98b8bd73d4c3f335e4cb346e6786e2d6de76d.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Harmonix Finance acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Harmonix Finance understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Harmonix Finance agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Harmonix Finance will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Harmonix Finance, the final report will be considered fully accepted by the Harmonix Finance without the signature.

2. AUDIT RESULT

2.1. Overview

The Rock Onyx Smart Contracts was written in `Solidity` language, with the required version to be `^0.8.19`.

Rock Onyx is a protocol designed to optimize user returns by providing automated Vaults with diverse investment strategies. The protocol consists of multiple contracts, and we will outline some of the core contracts below to shed light on its functionality.

2.1.1. RockOnyxEthLiquidityStrategy

A strategy contract aims to add liquidity to the `WETH/WSTETH` pair on the `Camelot` protocol to generate profit for the pair and earn rewards from the Camelot protocol. The contract utilizes `USDC` from user swaps, converting it to `WETH` and `WSTETH`, and then adds these tokens to a tick range set by the admin within the `RockOnyxUSDTVault` contract.

2.1.2. RockOynxUsdLiquidityStrategy

A strategy contract aims to add liquidity to the `USDC.e/USDC` pair on the `Camelot` protocol to generate profit for the pair and earn rewards from the `Camelot` protocol. The contract utilizes `USDC` from user swaps, converting it to `USDC.e`, and then adds these tokens to a tick range set by the admin within the `RockOnyxUSDTVault` contract.

2.1.3. RockOnyxOptionsStrategy

The OptionsStrategy contract utilizes `USDC` from user investments to interact with an optionsVendor designated by the admin. However, the current implementation lacks a clear definition for profit calculation within the code. As a result, the profit for each round is manually updated by the admin. Additionally, the contract's balance is managed through deposits made by a role called `ROCK_ONYX_OPTIONS_TRADER_ROLE`.

2.1.4. BaseRockOnyxOptionWheelVault

The protocol's base logic inherits functionality from the three strategies mentioned above. This contract extends the inherited logic to implement additional functions for managing the protocol. A crucial role within the contract is the `ROCK_ONYX_ADMIN_ROLE`, which grants the admin significant permissions. Notably, the admin can withdraw any tokens from the contract using the emergencyWithdraw function and modify the `VaultState` using the `importVaultState` function.

2.1.5. RockOnyxUSDTVault

The main contract, inheriting its logic from `BaseRockOnyxOptionWheelVault`, serves as the core of the protocol. It enables users to deposit `USDC`, which is then distributed across three investment strategies: `RockOnyxEthLiquidityStrategy`, `RockOynxUsdLiquidityStrategy`, and `RockOnyxOptionsStrategy`. Upon each deposit, users receive a corresponding number of shares.

The protocol manages user deposits and calculates profits periodically in distinct rounds. At the conclusion of each round, the protocol assesses all assets and profits generated by the strategies, subsequently increasing the share price. Users have the option to redeem their shares to obtain both their initial `USDC` deposit and any accrued profits.

To facilitate a smoother withdrawal process, the withdrawal logic is divided into two steps: `initialWithdraw` and `completeWithdraw`. After a round concludes, the protocol takes the accumulated profit and swaps a portion of the liquidity from the strategies into `USDC`. This enables users to complete their withdrawals during the `completeWithdraw` step.

2.2. Findings

During the audit process, the audit team found some vulnerability issue in the given version of Rock Onyx Smart Contracts.

#	Issue	Severity	Status
1	CamelotLiquidity.sol - Anyone can call `collectAllFess` to steal pool fees	CRITICAL	Fixed
2	BaseSwap.sol - Price manipulate attack in all logic using `getPriceOf` function	HIGH	Fixed
3	BaseSwap.sol - Missing the value of `amountOutMin` and `limitSqrtPrice` when swapping	HIGH	Acknowledged
4	RockOnyxUSDTVault.sol - Using wrong allocateRatio state in allocation function	HIGH	Fixed
5	RockOnyxUSDTVault.sol - Mistake in Performance Fee Calculation Formula.	HIGH	Fixed
6	RockOnyxUsdtVault.sol - Mistake when calculating pricePerShare	MEDIUM	Fixed

2.2.1. CamelotLiquidity.sol - Anyone can call `collectAllFess` to steal pool fees CRITICAL

To collect the fees from the pool, the LiquidityStrategy contracts approve their tokens for CamelotLiquidity to collect the fees. But the `collectAllFees` function in CamelotLiquidity allows anyone to call it and collect the fees from the pool. This can lead to a loss of funds for the project.

```
//RockOnyxEthLiquidityStrategy.sol
    function mintEthLPPosition(
        int24 lowerTick,
        int24 upperTick,
        uint16 ratio,
        uint8 decimals
    ) external nonReentrant {
        _auth(ROCK_ONYX_ADMIN_ROLE);
        ...
        IERC721(ethNftPositionAddress).approve(
            address(ethLPProvider),
            ethLPState.tokenId //@Verichains: Approve the token for CamelotLiquidity to collect the fees
        );
        ..
    }

    //CamelotLiquidity.sol
    function collectAllFees(
        uint tokenId
    ) external nonReentrant returns (uint256 amount0, uint256 amount1) {
        INonfungiblePositionManager.CollectParams
            memory params = INonfungiblePositionManager.CollectParams({
                tokenId: tokenId,
                recipient: msg.sender,//@Verichains: Anyone can call this function to collect the fees
                amount0Max: type(uint128).max,
                amount1Max: type(uint128).max
            });

        (amount0, amount1) = nonfungiblePositionManager.collect(params);
    }
```

UPDATES

- **Jun 06, 2024**: The issue has been acknowledged and fixed by the Harmonix Finance team.

2.2.2. BaseSwap.sol - Price manipulate attack in all logic using `getPriceOf` function HIGH

Lots of contracts in project use `getPriceOf` function to calculate the asset value. But the `getPriceOf` function gets the market price of token in uniswapV3 pool which can manipulate by attacker. This can lead to a loss of funds for the project.

```
function getPriceOf(
    address token0,
    address token1
) public view returns (uint256 price) {
    uint8 token0Decimals = ERC20(token0).decimals();
    uint8 token1Decimals = ERC20(token1).decimals();

    ISwapPool pool = ISwapPool(factory.poolByPair(token0, token1));
    address poolToken0 = pool.token0();
    (uint160 sqrtPriceX96, , , , , , , ) = pool.globalState(); //@Verichains: getPrice from
uniswapV3Pool which can be manipulated by attacker

    if (poolToken0 != token0)
        return 10 ** (token0Decimals + token1Decimals) / sqrtPriceX96ToPrice(sqrtPriceX96,
token1Decimals);

    return sqrtPriceX96ToPrice(sqrtPriceX96, token0Decimals);
}
```

RECOMMENDATION

It is recommended to use the Oracle price feed oracles to get the price of the token instead of using the `getPriceOf` function or using `TWAP` logic in UniswapV3.

UPDATES

- **Jun 06, 2024**: The issue has been acknowledged and fixed by the Harmonix Finance team.

2.2.3. BaseSwap.sol - Missing the value of `amountOutMin` and `limitSqrtPrice` when swapping HIGH

In the BaseSwap contract, the `swapTo` function performs token swaps with `amountOutmin` and `limitSqrtPrice` equal 0. It means that the function allows user swap with the current market price without any slippage protection. This allows an attacker to front-run the transaction and manipulate price, leading losses the tokenOut for users.

```
function swapTo(
        address recipient,
        address tokenIn,
        uint256 amountIn,
        address tokenOut
```

```
    ) external returns (uint256) {
        TransferHelper.safeTransferFrom(
            tokenIn,
            msg.sender,
            address(this),
            amountIn
        );
        TransferHelper.safeApprove(tokenIn, address(swapRouter), amountIn);

        ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
            .ExactInputSingleParams({
                tokenIn: tokenIn,
                tokenOut: tokenOut,
                recipient: recipient,
                deadline: block.timestamp,
                amountIn: amountIn,
                amountOutMinimum: 0, //@Verichains: missing amountOutMinimum
                limitSqrtPrice: 0 //@Verichains: missing limitSqrtPrice
            });

        return swapRouter.exactInputSingle(params);
    }
```

RECOMMENDATION

- It is recommended to add the value of `amountOutMin` and `limitSqrtPrice` to protect the user from the front-running attack.

UPDATES

The amountOutMinimum is calculated with slippage, yet it relies on market price (calculating in inside tx), which can still be influenced by price manipulation. Therefore, the logic of determining the getAmountOutMinimum doesn't resolve this issue.

The amountOutMinimum with slippage should be calculated in the client side and pass to the contract.

UPDATES

- **Jun 06, 2024**: The issue has been acknowledged by the Harmonix Finance team.

2.2.4. RockOnyxUSDTVault.sol - Using wrong allocateRatio state in allocation function HIGH

In allocateAssets function, the depositOptionsAmount is calculated using the usdLPRatio instead of optionsRatio. This is an issue as it can lead to an incorrect allocation of assets in the vault. This can lead to a loss of funds for the users.

```
function allocateAssets() private {
        uint256 depositToEthLPAmount = vaultState.pendingDepositAmount *
allocateRatio.ethLPRatio / 10 ** allocateRatio.decimals;
        uint256 depositToUsdLPAmount = vaultState.pendingDepositAmount *
allocateRatio.usdLPRatio / 10 ** allocateRatio.decimals;
        uint256 depositOptionsAmount = vaultState.pendingDepositAmount *
allocateRatio.usdLPRatio / 10 ** allocateRatio.decimals; //@Verichains: This should be
allocateRatio.optionsRatio
        vaultState.pendingDepositAmount -= (depositToEthLPAmount + depositToUsdLPAmount +
depositOptionsAmount);

        depositToEthLiquidityStrategy(depositToEthLPAmount);
        depositToUsdLiquidityStrategy(depositToUsdLPAmount);
        depositToOptionsStrategy(depositOptionsAmount);
    }
```

RECOMMENDATION

Change `allocateRatio.usdLPRatio` in depositOptionAmount calculating statement to `allocateRatio.optionsRatio`.

UPDATES

- **Jun 06, 2024**: The issue has been acknowledged and fixed by the Harmonix Finance team.

2.2.5. RockOnyxUSDTVault.sol - Mistake in Performance Fee Calculation Formula. HIGH

The division operation takes precedence over the addition operation. In the `completeWithdrawl` function, there is a mistake in the sequence of operations for calculating the performance fee. The part `withdrawals[msg.sender].shares / withdrawals[msg.sender].shares + depositReceipt.shares` will be `1 + depositReceipt.shares`. This formula does not correspond to the `depositReceipt.depositAmount` formula in the same function.

```
function completeWithdrawal(uint256 shares) external nonReentrant {
        require(withdrawals[msg.sender].shares >= shares, "INVALID_SHARES");
        require(vaultState.withdrawPoolAmount > 0, "EXCEED_WITHDRAW_POOL_CAPACITY");

        DepositReceipt storage depositReceipt = depositReceipts[msg.sender];

        uint256 withdrawAmount = ShareMath.sharesToAsset(
            shares,
            roundPricePerShares[currentRound-1],
            vaultParams.decimals
        );
```

```
        (uint256 profit,) = getPnL();

        uint256 performanceFee = profit > 0 ?
            (profit * depositReceipt.depositAmount) * (withdrawals[msg.sender].shares /
withdrawals[msg.sender].shares + depositReceipt.shares) * vaultParams.performanceFeeRate /
1e8 : 0; //@Verichains: This should be (withdrawals[msg.sender].shares /
(withdrawals[msg.sender].shares + depositReceipt.shares))

        vaultState.performanceFeeAmount += performanceFee;
        withdrawAmount -= (performanceFee + NETWORK_COST);
        vaultState.withdrawPoolAmount -=  withdrawAmount;
        depositReceipt.depositAmount -= shares * depositReceipt.depositAmount /
(depositReceipt.shares + withdrawals[msg.sender].shares);
        withdrawals[msg.sender].shares -= shares;

        IERC20(vaultParams.asset).safeTransfer(msg.sender, withdrawAmount);

        emit Withdrawn(msg.sender, withdrawAmount, withdrawals[msg.sender].shares);
    }
```

RECOMMENDATION

The part `withdrawals[msg.sender].shares / withdrawals[msg.sender].shares + depositReceipt.shares` should be `(withdrawals[msg.sender].shares / (withdrawals[msg.sender].shares + depositReceipt.shares))`.

UPDATES

- **Jun 06, 2024**: The issue has been acknowledged and fixed by the Harmonix Finance team.

2.2.6. RockOnyxUsdtVault.sol - Mistake when calculating pricePerShare MEDIUM

When the admin calls the `closeRound` function, the `pricePerShare` is calculated based on the totalValueLocked and totalShares. However, the two-step withdrawal logic in the contract leads to an error in calculating the pricePerShare.

The withdrawal logic consists of two steps: `initialWithdraw` and `completeWithdraw`. The totalShares is not subtracted during this process, but the `totalValueLocked` decreases after the second step is completed.

Consequently, the `pricePerShare` calculation in the `closeRound` function is incorrect. Since the calculation uses the larger, current value of totalShares, the resulting `pricePerShare` is lower than the actual value.

```
function closeRound() external nonReentrant {
        _auth(ROCK_ONYX_ADMIN_ROLE);
```

```
        closeEthLPRound();
        closeUsdLPRound();
        closeOptionsRound();
        vaultState.currentRoundFeeAmount = getManagementFee();
        roundPricePerShares[currentRound] = ShareMath.pricePerShare(
            vaultState.totalShares, // @Verichains: totalShares not subtracted after
withdraw done
            _totalValueLocked() - vaultState.currentRoundFeeAmount, // @Verichains:
totalValueLocked was subtract withdrawAmount when user complete withdraw in step 2
            vaultParams.decimals
        );
        vaultState.totalShares -= roundWithdrawalShares[currentRound]; // @Verichains:
totalShares should be subtracted by the user share withdraw after the withdraw process is
done
        recalculateAllocateRatio();
        emit RoundClosed(
            currentRound,
            _totalValueLocked(),
            vaultState.currentRoundFeeAmount
        );
        currentRound++;
    }
```

The totalShares should be subtracted by the user's withdrawn share after the withdrawal process is completed. This will ensure the pricePerShare is calculated correctly.

UPDATES

- **Jun 06, 2024**: The issue has been acknowledged and fixed by the Harmonix Finance team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jun 06, 2024*	Public Report	Verichains Lab

Table 2. Report versions history