# SMART CONTRACT AUDIT REPORT

for

# ConvexStakingWrapperOhmSync

Prepared By: Xiaomi Huang

PeckShield

September 15, 2022

## Document Properties

| | |
|---|---|
| Client | Convex Finance |
| Title | Smart Contract Audit Report |
| Target | ConvexStakingWrapperOhmSync Contract |
| Version | 1.0 |
| Author | Luck Hu |
| Auditors | Luck Hu, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | September 15, 2022 | Luck Hu | Final Release |
| 1.0-rc | September 12, 2022 | Luck Hu | Release Candidate |

## 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 ConvexStakingWrapperOhmSync contract, 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 ConvexStakingWrapperOhmSync Contract

The ConvexStakingWrapperOhmSync contract is a tokenized wrapper for staking on Convex for a given pool. It allows an outside mechanism to donate LP tokens to all staked users by using a sync() function that can increase the amount of assets (LP tokens) each share can claim. The basic information of the audited smart contract is as follows:

Table 1.1: Basic Information of ConvexStakingWrapperOhmSync Contract

| Item | Description |
|---:|---|
| Target | ConvexStakingWrapperOhmSync Contract |
| Website | https://www.convexfinance.com/ |
| Type | Smart Contract |
| Language | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | September 15, 2022 |

In the following, we show the Git repository of reviewed file and the commit hash value used in this audit. Note the audit scope only covers the contracts/contracts/wrappers/ConvexStakingWrapperOhmSync .sol contract of the main branch.

- https://github.com/convex-eth/platform.git (9adc57e)

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

- https://github.com/convex-eth/platform.git (9933dfd)

## 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 |



## 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 classified into four categories accordingly, 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

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 |

contract is considered safe regarding the check item. For any discovered issue, we might further 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.

## 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 `ConvexStakingWrapperOhmSync` implementation. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

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 that 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, the smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key ConvexStakingWrapperOhmSync Contract Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Medium | Sybil Attacks to Drain Vault Rewards | Business Logic | Fixed |
| PVE-002 | Low | Non ERC4626-Compliance of ConvexStakingWrapperOhmSync | Coding Practices | Fixed |
| PVE-003 | Low | Trust Issue of Admin Keys | Security Features | Mitigated |

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 Sybil Attacks to Drain Vault Rewards

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: ConvexStakingWrapperOhmSync
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

The ConvexStakingWrapperOhmSync contract provides an interface, i.e., shutdown(), that allows the privileged owner to shut down the contract. Once the contract is shut down, user deposit is prohibited and normal checkpointing is not allowed. The purpose here is to ensure users can always have the ability to withdraw their deposits.

To elaborate, we show below the code snippets from the ConvexStakingWrapperOhmSync contract. When the contract is shut down, normal checkpointing is directly returned in the _checkpoint() routine (line 400). Hence users rewards are not settled before the token transfers. However, users can update the checkpoints and claim their rewards via the _checkpointAndClaim() routine, even after the contract is shut down. As a result, the current logic is vulnerable to so-called Sybil attacks to drain all rewards in the wrapper.

Let's assume at the very beginning there is a malicious actor named Malice, who owns 100 share tokens. Malice has an accomplice named Trudy who currently has 0 balance of the wrapper share. This Sybil attack can be launched as follows:

```
    function shutdown() external onlyOwner {
        isShutdown = true;
    }
```

Listing 3.1: shutdown()

```
398     function _checkpoint(address[2] memory _accounts) internal nonReentrant{
399         //if shutdown, no longer checkpoint in case there are problems
```

```
        if(isShutdown) return;

        uint256 supply = _getTotalSupply();
        uint256[2] memory depositedBalance;
        depositedBalance[0] = _getDepositedBalance(_accounts[0]);
        depositedBalance[1] = _getDepositedBalance(_accounts[1]);

        IRewardStaking(convexPool).getReward(address(this), true);

        uint256 rewardCount = rewards.length;
        for (uint256 i = 0; i < rewardCount; i++) {
           _calcRewardIntegral(i,_accounts,depositedBalance,supply,false);
        }
    }

    function _checkpointAndClaim(address[2] memory _accounts) internal nonReentrant{

        uint256 supply = _getTotalSupply();
        uint256[2] memory depositedBalance;
        depositedBalance[0] = _getDepositedBalance(_accounts[0]); //only do first slot

        IRewardStaking(convexPool).getReward(address(this), true);

        uint256 rewardCount = rewards.length;
        for (uint256 i = 0; i < rewardCount; i++) {
           _calcRewardIntegral(i,_accounts,depositedBalance,supply,true);
        }
    }
```

Listing 3.2: `ConvexStakingWrapperOhmSync.sol`

```
    function _beforeTokenTransfer(address _from, address _to, uint256 _amount) internal
        override {
        _checkpoint([_from, _to]);
    }
```

Listing 3.3: `_beforeTokenTransfer()`

1. `Malice` initially claims his rewards and then transfers 100 shares to `Trudy` (or `M_1`), who can now claim the rewards one more time!

2. `M_1` claims the rewards and then transfers 100 shares to `M_2`, who can also claim the rewards one more time.

3. We can repeat by transferring $M_i$'s 100 shares balance to $M_{i+1}$ who can also claim the rewards. In other words, we can effectively drain all rewards with new accounts created and iterated!

**Recommendation** To mitigate, it is necessary to settle users rewards before token transfers, or forbid rewards claiming after the contract is shutdown. By doing so, we can effectively mitigate the above `Sybil` attacks.

**Status** This issue has been fixed by this commit: `c9fcf4c`.

## 3.2 Non ERC4626-Compliance of ConvexStakingWrapperOhmSync

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ConvexStakingWrapperOhmSync`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

### Description

The `ConvexStakingWrapperOhmSync` contract is a tokenized wrapper for staking on `Convex` for a given pool. It is also designed to be compliant with the `ERC4626` specification. While reviewing the implementation of the interfaces defined in the `IERC4626`, we notice current contract is not fully compliant with `ERC4626`.

Firstly, the `EIP-4626` defines a `maxDeposit()` interface which is specified that "MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0". We notice that if the contract is shutdown, the deposit/mint are disabled. However, the `maxDeposit()` routine does not return 0 for this case which needs to be corrected. Note same issue exists in the `maxMint()` routine.

```
    function maxDeposit(address _receiver) external override view returns (uint256){
        return uint256(-1);
    }
    function maxMint(address _receiver) external override view returns (uint256){
        return uint256(-1);
    }
```

Listing 3.4: `maxDeposit()/maxMint()`

Secondly, the `EIP-4626` defines a `previewDeposit()` interface which is specified that "MUST return as close to and no more than the exact amount of Vault shares that would be minted in a deposit call in the same transaction". So the `previewDeposit()` routine is expected to return the amount of `Vault` shares to be minted from a deposit of the desired amount of assets. However, current implementation returns the amount of assets from a call to the `convertToAssets()` (line 177). Our analysis shows that it shall call the `convertToShares()` routine instead. Similarly, the `previewMint()` routine is expected to return the amount of assets that need to be deposited to receive the desired amount of shares. The call to the `convertToShares()` (line 180) shall be corrected to the `convertToAssets()`.

What is more, in `EIP-4626`, the `previewMint()` is specified to "MUST return as close to and no fewer than the exact amount of assets that would be deposited in a mint call". So the `previewMint()` routine shall return a rounding up value from the assets amount calculation. However, current implementation returns a rounding down value. Similarly, the `previewWithdraw()` routine is expected to return a rounding up value from the shares amount calculation. However, current implementation returns a rounding down value (line 183).

```
176     function previewDeposit(uint256 _amount) external override view returns (uint256){
177         return convertToAssets(_amount);
178     }
179     function previewMint(uint256 _shares) external override view returns (uint256){
180         return convertToShares(_shares);
181     }
182     function previewWithdraw(uint256 _amount) external override view returns (uint256){
183         return convertToShares(_amount);
184     }
```

Listing 3.5: `previewDeposit()/previewMint()/previewWithdraw()`

Thirdly, the `EIP-4626` defines a `maxWithdraw()` interface which is specified to return the "Maximum amount of the underlying asset that can be withdrawn from the owner balance in the Vault". However, current implementation directly returns `uint256(-1)` (line 183) which is `2 ** 256 - 1`. Our analysis shows that the `maxWithdraw()` implementation can be corrected to `convertToAssets(balanceOf(_owner))`. Similarly, the `maxRedeem()` interface in `EIP-4626` is specified to return the "Maximum amount of Vault shares that can be redeemed from the owner balance in the Vault". However, current implementation directly returns `uint256(-1)` (line 189) which is `2 ** 256 - 1`. Our analysis shows that it shall return `balanceOf(_owner)`.

```
182     function maxWithdraw(address _owner) external override view returns (uint256){
183         return uint256(-1);
184     }
```

Listing 3.6: `maxWithdraw()`

```
188     function maxRedeem(address _owner) external override view returns (uint256){
189         return uint256(-1);
190     }
```

Listing 3.7: `maxRedeem()`

Lastly, the `mint()` routine is specified to mint exactly `_shares` amount of shares to the receiver by depositing `assets` of the underlying tokens. Current implementation calls the `convertToAssets()` (line 202) to calculate the required assets amount which is a rounding down value that may introduce slippage in share price. Our study shows that it shall use the `previewMint()` routine instead. Similarly, the `withdraw()` routine shall use the `previewWithdraw()` routine to compute the required shares that need to be burned to receive the desired amount of assets.

```
    function mint(uint256 _shares, address _receiver) external override returns (uint256
         assets){
        require(!isShutdown, "shutdown");

        //dont need to call checkpoint since _mint() will

        if (_shares > 0) {
            assets = convertToAssets(_shares);
            if(assets > 0){
                _mint(_receiver, _shares);
                IERC20(curveToken).safeTransferFrom(msg.sender, address(this), assets);
                IConvexDeposits(convexBooster).deposit(convexPoolId, assets, true);
                emit Deposit(msg.sender, _receiver, assets, _shares);
            }
        }
    }
```

Listing 3.8: `mint()`

```
    function withdraw(uint256 _amount, address _receiver, address _owner) public
        override returns(uint256 shares){

        //dont need to call checkpoint since _burn() will

        if (_amount > 0) {
            shares = convertToShares(_amount);//1Luck: previewWithdraw(_amount);
            _burn(msg.sender, shares);
            IRewardStaking(convexPool).withdrawAndUnwrap(_amount, false);
            IERC20(curveToken).safeTransfer(_receiver, _amount);
            emit Withdraw(msg.sender, _receiver, msg.sender, shares, _amount);
        }
    }
```

Listing 3.9: `withdraw()`

**Recommendation** Revise the above mentioned functions to ensure their `EIP-4626` compliance.

**Status** This issue has been fixed by this commit: `c9fcf4c`.

## 3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ConvexStakingWrapperOhmSync
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

### Description

In the `ConvexStakingWrapperOhmSync` contact, there is a privileged account, i.e., `owner`, that is privileged to shut down the contract. Once the contract is shut down, the deposit, mint and stake operations are disabled, and users rewards claiming may also be impacted.

```
    function shutdown() external onlyOwner {
        isShutdown = true;
    }
```

Listing 3.10: `shutdown()`

```
function _checkpoint(address[2] memory _accounts) internal nonReentrant{
    //if shutdown, no longer checkpoint in case there are problems
    if(isShutdown) return;

    uint256 supply = _getTotalSupply();
    uint256[2] memory depositedBalance;
    depositedBalance[0] = _getDepositedBalance(_accounts[0]);
    depositedBalance[1] = _getDepositedBalance(_accounts[1]);

    IRewardStaking(convexPool).getReward(address(this), true);

    uint256 rewardCount = rewards.length;
    for (uint256 i = 0; i < rewardCount; i++) {
       _calcRewardIntegral(i,_accounts,depositedBalance,supply,false);
    }
}
```

Listing 3.11: `_checkpoint()`

We understand the need of the privileged function for emergency, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

**Recommendation** Promptly transfer the privileged account to the intended `DAO`-like governance contract. The change to the privileged operation 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** This issue has been mitigated as the team confirmed they plan to use multi-sig.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the `ConvexStakingWrapperOhmSync` contract, which is a tokenized wrapper for staking on `Convex` for a given pool. It allows an outside mechanism to donate `LP` tokens to all staked users by using a `sync()` function that can increase the amount of assets (`LP` tokens) each share can claim. The current code base is well structured and neatly organized. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.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.