



# SMART CONTRACT AUDIT REPORT

for

## Convex-Eth Sidechain



Prepared By: Xiaomi Huang

PeckShield  
November 9, 2022

## Document Properties

Client	Convex Finance
Title	Smart Contract Audit Report
Target	Convex-Eth Sidechain
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	November 9, 2022	Xiaotao Wu	Final Release
1.0-rc	October 23, 2022	Xiaotao Wu	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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Convex-Eth Sidechain . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Redundant Code Removal . . . . .	11
3.2	Accommodation Of Non-ERC20-Compliant Tokens . . . . .	12
3.3	Trust Issue of Admin Keys . . . . .	14
<b>4</b>	<b>Conclusion</b>	<b>16</b>
	<b>References</b>	<b>17</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Convex-Eth Sidechain feature, 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 Convex-Eth Sidechain

Convex Finance is a platform built to boost rewards for CRV stakers and liquidity providers alike, all in a simple and easy to use interface. Convex aims to simplify staking on Curve, as well as the CRV-locking system with the help of its native fee-earning token: CVX. The audited Convex-Eth Sidechain feature is designed to support staking on non-Ethereum mainnet chains. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Convex-Eth Sidechain

Item	Description
Name	Convex Finance
Website	<a href="https://www.convexfinance.com/">https://www.convexfinance.com/</a>
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 9, 2022

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

- <https://github.com/convex-eth/sidechain-platform> (eed1e14)

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

- <https://github.com/convex-eth/sidechain-platform> (2c50d49)

## 1.2 About PeckShield

PeckShield Inc. [8] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- 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) [6], 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
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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 `Convex-Eth Sidechain` 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 logics, 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	0	
Low	2	
Informational	1	
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, 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 2 low-severity vulnerabilities and 1 informational suggestion.

Table 2.1: Key Convex-Eth Sidechain Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Redundant Code Removal	Coding Practices	Resolved
PVE-002	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

## 3 | Detailed Results

### 3.1 Redundant Code Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ConvexRewardPool
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

#### Description

In Convex-Eth Sidechain feature, the `ConvexRewardPool::stakeFor()` function will be called by the Booster contract to stake on behalf of the LP depositor.

While reviewing the implementation of this routine, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. Specifically, the checkpoint logic is executed before minting Convex Deposit tokens for the depositor (line 319). But the same code is executed in the `_beforeTokenTransfer()` function, which will be invoked by the `_mint()` function in line 323. Therefore, we suggest to remove this redundant checkpoint logic in line 319.

In addition, we note that the `_beforeTokenTransfer()` function can be improved by only executing the checkpoint logic for account that not equal to `address(0)`.

```
312 //deposit/stake on behalf of another account
313 function stakeFor(address _for, uint256 _amount) external nonReentrant returns(bool)
314 {
315     require(msg.sender == convexBooster, "!auth");
316     require(_amount > 0, 'RewardPool : Cannot stake 0');
317
318     //checkpoint first
319     _checkpoint(_for);
320
321     //change state
322     //assign to _for
```

```

323     _mint(_for, _amount);
324
325     emit Staked(_for, _amount);
326
327     return true;
328 }

```

Listing 3.1: ConvexRewardPool::stakeFor()

```

354     function _beforeTokenTransfer(address _from, address _to, uint256 _amount) internal
        override {
355         _checkpoint(_from);
356         _checkpoint(_to);
357     }

```

Listing 3.2: ConvexRewardPool::\_beforeTokenTransfer()

**Recommendation** Consider the removal of the redundant code with a simplified, consistent implementation.

**Status** This issue has been fixed in the commit: [2c50d49](#).

## 3.2 Accommodation Of Non-ERC20-Compliant Tokens

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

### Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!(_value != 0) && (allowed[msg.sender][_spender] != 0))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (<https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729>).

```

194     /**

```

```

195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201         // To change the approve amount you first have to reduce the addresses'
202         // allowance to zero by calling 'approve(_spender, 0)' if it is not
203         // already 0 to mitigate the race condition described here:
204         // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }

```

Listing 3.3: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount) external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of USDT's `approve()`, the call will be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In the following, we use the `VoterProxy::deposit()` routine as an example. If the USDT token is supported as `_token`, the unsafe version of `IERC20(_token).approve(_gauge, _amount)` may revert as there is no return value in the USDT token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```

54     function deposit(address _token, address _gauge, uint256 _amount) external returns (
        bool){
55         require(msg.sender == operator, "!auth");
56         if(protectedTokens[_token] == false){
57             protectedTokens[_token] = true;
58         }
59         if(protectedTokens[_gauge] == false){
60             protectedTokens[_gauge] = true;
61         }
62         // uint256 balance = IERC20(_token).balanceOf(address(this));
63         if (_amount > 0) {

```

```

64         IERC20(_token).approve(_gauge, _amount);
65         IGauge(_gauge).deposit(_amount);
66     }
67     return true;
68 }

```

Listing 3.4: VoterProxy::deposit()

**Recommendation** Accommodate the above-mentioned idiosyncrasy about ERC20-related approve().

**Status** This issue has been fixed in the commit: [b36c848](#).

### 3.3 Trust Issue of Admin Keys

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

#### Description

In Convex-Eth Sidechain feature, there is a privileged account, i.e., `owner`. This account play a critical role in governing and regulating the system-wide operations (e.g., set CRV address for a given Curve pool factory, set `rescueManager`, set `rewardManager`, set factories for reward and deposit token contracts, set `fees`, shutdown the system, set key parameters for the `RewardManager/RewardFactory/TokenFactory` contracts, set `operator` for the `VoterProxy` contract, etc.). Our analysis shows that this privileged account need to be scrutinized. In the following, we use the `VoterProxy` contract as an example and show the representative function potentially affected by the privileges of the `owner` account.

```

47     function setOperator(address _operator) external {
48         require(msg.sender == owner, "!auth");
49         require(operator == address(0) || IBooster(operator).isShutdown() == true, "needs
           shutdown");
50
51         operator = _operator;
52     }

```

Listing 3.5: Example Privileged Operation in VoterProxy

We emphasize that the privilege assignments are necessary and required for various protocol maintenance operations. The user's deposited assets are not affected by these privileges. But we should point out that 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. All changes 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** This issue has been mitigated as the team confirms that multi-sig will be adopted for the privileged account.



## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Convex-Eth Sidechain` feature. The audited `Convex-Eth Sidechain` feature is the `Convex` smart contract platform for staking on `Curve` on non-`Ethereum` mainnet chains. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based 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-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.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 VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [7] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [8] PeckShield. PeckShield Inc. <https://www.peckshield.com>.