



BlockSec

Security Audit Report for Coordinape Protocol

Date: April 11, 2022

Version: 1.1

Contact: contact@blocksecteam.com

Contents

1	Introduction	1
1.1	About Target Contracts	1
1.2	Disclaimer	1
1.3	Procedure of Auditing	1
1.3.1	Software Security	2
1.3.2	DeFi Security	2
1.3.3	NFT Security	2
1.3.4	Additional Recommendation	2
1.4	Security Model	3
2	Findings	4
2.1	Software Security	4
2.1.1	Incorrect delay time in schedule	4
2.1.2	Potential unsafe check in TimeLock	5
2.2	DeFi Security	5
2.2.1	Insufficient check for <code>_token</code> and <code>_simpleToken</code>	5
2.2.2	Lack of authorization of the <code>apeWithdrawSimpleToken</code> function	6
2.2.3	Insufficient sanitation in <code>uploadEpochRoot</code>	6
2.3	Additional Recommendation	8
2.3.1	Do not use unlimited approval	8
2.3.2	Use <code>safeTransfer</code>	9
2.3.3	Ensure that the same storage layout of the proxy	10
2.3.4	Ensure that state variables in <code>ApeRegistry</code> must be properly initialized	10
2.3.5	Remove unused variables in <code>ApeVault.sol</code>	11

Report Manifest

Item	Description
Client	Coordinape
Target	Coordinape Protocol

Version History

Version	Date	Description
1.1	April 11, 2022	Update the status for issue 5 described in Section 2.2.3
1.0	March 20, 2022	First Release

About BlockSec The **BlockSec Team** focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at [Email](#), [Twitter](#) and [Medium](#).

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The repository that has been audited includes `coordinape-protocol` ¹. The auditing process is iterative. Specifically, we first audit the code in Version 1 and then audit the commits that fix the issues raised in Version 1. If there exist new issues, we will continue this process. The commit SHA values during the audit are shown in the following.

Project		Commit SHA
Coordinape	Version 1	7a8e6173305696c72195fa4242126d284611270c
	Version 2	de2c00ad3421d7c72f2425c83a9eb950d03d81da

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.

¹<https://github.com/coordinape/coordinape-protocol/>

- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.
We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style



Note The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology ² and Common Weakness Enumeration ³. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact	High	High	Medium
	Low	Medium	Low
		High	Low
		Likelihood	

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

²https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

³<https://cwe.mitre.org/>

Chapter 2 Findings

In total, we find **five** potential issues in the smart contract. We also have **five** recommendations, as follows:

- High Risk: 2
- Medium Risk: 2
- Low Risk: 1
- Recommendations: 5

ID	Severity	Description	Category	Status
1	Medium	<i>Incorrect delay time in schedule</i>	Software Security	Fixed
2	Medium	<i>Potential unsafe check in TimeLock</i>	Software Security	Fixed
3	Low	<i>Insufficient check for <code>_token</code> and <code>_simpleToken</code></i>	DeFi Security	Fixed
4	High	<i>Lack of authorization of the <code>apeWithdrawSimpleToken</code> function</i>	DeFi Security	Fixed
5	High	<i>Insufficient sanitation in <code>uploadEpochRoot</code></i>	DeFi Security	Fixed
6	-	<i>Do not use unlimited approval</i>	Recommendation	Fixed
7	-	<i>Use <code>safeTransfer</code></i>	Recommendation	Fixed
8	-	<i>Ensure that the same storage layout of the proxy</i>	Recommendation	Acknowledged
9	-	<i>Ensure that state variables in <code>ApeRegistry</code> must be properly initialized</i>	Recommendation	Fixed
10	-	<i>Remove unused variables in <code>ApeVault.sol</code></i>	Recommendation	Fixed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Incorrect delay time in schedule

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description The `schedule` function in the `TimeLock` has an implementation error that the `minDelay` on line 58 should be `_delay`. This can make the `TimeLock` mechanism work unexpectedly, since the task will be scheduled immediately, instead of with the expected delay (`_delay`).

```
54 function schedule(address _target, bytes calldata _data, bytes32 _predecessor, bytes32 _salt,
    uint256 _delay) external onlyOwner {
55     bytes32 id = hashOperation(_target, _data, _predecessor, _salt);
56     require(timestamps[id] == 0, "TimeLock: Call already scheduled");
57     require(_delay >= minDelay, "TimeLock: Insufficient delay");
58     timestamps[id] = block.timestamp + minDelay;
59     emit CallScheduled(id, _target, _data, _predecessor, _delay);
60 }
```

Listing 2.1: `TimeLock.sol`

Impact Make the `TimeLock` mechanism work unexpectedly.

Suggestion Use `_delay` instead of `minDelay` on line 58.

2.1.2 Potential unsafe check in TimeLock

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description If `timestamps[id]` is accidentally to be set as 0, then the require check of the `execute()` function (line 70 - line 71) could be bypassed. Actually this sanity check is inconsistent with the logic `isOperationReady` in OpenZeppelin.

```
46 function isDoneCall(bytes32 _id) public view returns(bool) {
47     return timestamps[_id] == _DONE_TIMESTAMP;
48 }
49
50 function isReadyCall(bytes32 _id) public view returns(bool) {
51     return timestamps[_id] <= block.timestamp;
52 }
```

Listing 2.2: TimeLock.sol

```
68 function execute(address _target, bytes calldata _data, bytes32 _predecessor, bytes32 _salt,
    uint256 _delay) external onlyOwner {
69     bytes32 id = hashOperation(_target, _data, _predecessor, _salt);
70     require(isReadyCall(id), "TimeLock: Not ready for execution");
71     require(!isDoneCall(id), "TimeLock: Already executed");
72     require(_predecessor == bytes32(0) || isDoneCall(_predecessor), "TimeLock: Predecessor call
    not executed");
73     _call(id, _target, _data);
74     timestamps[id] = _DONE_TIMESTAMP;
75 }
```

Listing 2.3: TimeLock.sol

```
138 function isOperationReady(bytes32 id) public view virtual returns (bool ready) {
139     uint256 timestamp = getTimestamp(id);
140     return timestamp > _DONE_TIMESTAMP && timestamp <= block.timestamp;
141 }
```

Listing 2.4: TimelockController.sol in OpenZeppelin

Impact The sanity check in `execute` could be bypassed.

Suggestion Make it consistent with OpenZeppelin.

2.2 DeFi Security

2.2.1 Insufficient check for `_token` and `_simpleToken`

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description

According to the document, `_token` and `_simpleToken` cannot be zero at the same time. However, there does not exist such a verification in the `init()` function.

```
72 function init(  
73     address _apeRegistry,  
74     address _token,  
75     address _registry,  
76     address _simpleToken,  
77     address _newOwner) external {  
78     require(!setup);  
79     setup = true;  
80     apeRegistry = _apeRegistry;  
81     if (_token != address(0))  
82         vault = VaultAPI(RegistryAPI(_registry).latestVault(_token));  
83     simpleToken = IERC20(_simpleToken);  
84  
85     // Recommended to use a token with a 'Registry.latestVault(_token) != address(0)'  
86     token = IERC20(_token);  
87     // Recommended to use 'v2.registry.ychad.eth'  
88     registry = RegistryAPI(_registry);  
89     _owner = _newOwner;  
90     emit OwnershipTransferred(address(0), _newOwner);  
91 }
```

Listing 2.5: wrapper/beacon/ApeVault.sol

Impact NA

Suggestion Add a check in the function.

2.2.2 Lack of authorization of the `apeWithdrawSimpleToken` function

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description

The public function `apeWithdrawSimpleToken` lacks a modifier so that everyone can withdraw the simple token from the contract.

```
131 function apeWithdrawSimpleToken(uint256 _amount) public {  
132     simpleToken.safeTransfer(msg.sender, _amount);  
133 }
```

Listing 2.6: wrapper/beacon/ApeVault.sol

Impact Anyone can withdraw the simple token in the contract.

Suggestion Add the `onlyOwner` modifier.

2.2.3 Insufficient sanitation in `uploadEpochRoot`

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description

The `uploadEpochRoot` is used to uploadEpochRoot that can be used to claim the tokens in each epoch. However, we find that this function does not have enough sanitation that could be abused by attackers to claim tokens.

The possible attack process is as following:

1. The attacker invokes `uploadEpochRoot`, with a `_vault` contract that is controlled (deployed) by the attacker. The passed `_token` is a valid token in the distributor contract. The passed `_amount` is zero. There is no check on the passing `_vault` parameter. Since `_vault` is controlled by the attacker, the attacker can control the return value of the function call to this contract. Specifically, the attacker returns `msg.sender` to the invocation `_vault.owner()`. Thus, `isOwner` is true on line 60. So the check on line 61 can pass. Similarly, the require statement on line 62 can be bypassed since the return value of `_vault.vault()` is controlled by the attacker. Then from line 65-69, the code will update the `epochRoots` to the `_root` passed by the attacker. The check from line 73 does not help here since the passed `_amount` is zero.
2. Then the attacker can invoke `claim` function. Since the `_root` has been updated in the previous step, the check on line 118 can pass and the attacker can drain the tokens in the circle.

```
52 function uploadEpochRoot(  
53     address _vault,  
54     bytes32 _circle,  
55     address _token,  
56     bytes32 _root,  
57     uint256 _amount,  
58     uint8 _tapType)  
59 external {  
60     bool isOwner = ApeVaultWrapperImplementation(_vault).owner() == msg.sender;  
61     require(vaultApprovals[_vault][_circle] == msg.sender || isOwner, "Sender cannot upload a root  
        ");  
62     require(address(ApeVaultWrapperImplementation(_vault).vault()) == _token, "Vault cannot supply  
        token");  
63     if (!isOwner)  
64         _isTapAllowed(_vault, _circle, _token, _amount);  
65     uint256 epoch = epochTracking[_circle][_token];  
66     epochRoots[_circle][_token][epoch] = _root;  
67  
68     epochTracking[_circle][_token]++;  
69     circleAlloc[_circle][_token] += _amount;  
70     uint256 beforeBal = IERC20(_token).balanceOf(address(this));  
71     uint256 sharesRemoved = ApeVaultWrapperImplementation(_vault).tap(_amount, _tapType);  
72     uint256 afterBal = IERC20(_token).balanceOf(address(this));  
73     require(afterBal - beforeBal == _amount, "Did not receive correct amount of tokens");  
74     if (sharesRemoved > 0)  
75         emit apeVaultFundsTapped(_vault, address(ApeVaultWrapperImplementation(_vault).vault()),  
            sharesRemoved);  
76 }
```

Listing 2.7: ApeDistributor.sol

```
115 function claim(bytes32 _circle, address _token, uint256 _epoch, uint256 _index, address _account  
    , uint256 _checkpoint, bool _redeemShares, bytes32[] memory _proof) public {  
116     require(!isClaimed(_circle, _token, _epoch, _index), "Claimed already");
```

```
117 bytes32 node = keccak256(abi.encodePacked(_index, _account, _checkpoint));
118 require(_proof.verify(epochRoots[_circle][_token][_epoch], node), "Wrong proof");
119 uint256 currentCheckpoint = checkpoints[_circle][_token][_account];
120 require(_checkpoint > currentCheckpoint, "Given checkpoint not higher than current checkpoint"
    );
121
122 uint256 claimable = _checkpoint - currentCheckpoint;
123 require(claimable <= circleAlloc[_circle][_token], "Can't claim more than circle has to give")
    ;
124 circleAlloc[_circle][_token] -= claimable;
125 checkpoints[_circle][_token][_account] = _checkpoint;
126 _setClaimed(_circle, _token, _epoch, _index);
127 if (_redeemShares && msg.sender == _account)
128     VaultAPI(_token).withdraw(claimable, _account);
129 else
130     IERC20(_token).safeTransfer(_account, claimable);
131 emit Claimed(_circle, _token, _epoch, _index, _account, claimable);
132 }
```

Listing 2.8: ApeDistributor.sol

Impact The attacker can update the EpochRoot as the vault owner.

Suggestion Ensure that the passing `_vault` is a valid one (not a fake one deployed by the attacker.)

Note The project fixes this issue by using `_vault` as the first index of `epochRoots`. In this case, even the attacker can pass a fake `_vaule`, it will not cause the security impact since it only updates the epochRoot for that fake value.

2.3 Additional Recommendation

2.3.1 Do not use unlimited approval

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description

The `delegateDeposit` use unlimited approval, which is controversial and widely debated in recent months due to some relevant security incidents.

```
43 function delegateDeposit(address _apeVault, address _token, uint256 _amount) external returns(
    uint256 deposited) {
44     VaultAPI vault = VaultAPI(RegistryAPI(yearnRegistry).latestVault(_token));
45     require(address(vault) != address(0), "ApeRouter: No vault for token");
46     require(ApeVaultFactoryBeacon(apeVaultFactory).vaultRegistry(_apeVault), "ApeRouter: Vault does
        not exist");
47     require(address(vault) == address(ApeVaultWrapperImplementation(_apeVault).vault()), "ApeRouter:
        yearn Vault not identical");
48
49     IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
50
51     if (IERC20(_token).allowance(address(this), address(vault)) < _amount) {
```

```
52     IERC20(_token).safeApprove(address(vault), 0); // Avoid issues with some IERC20(_token)s
        requiring 0
53     IERC20(_token).safeApprove(address(vault), MAX_UINT); // Vaults are trusted
54 }
55
56 uint256 beforeBal = IERC20(_token).balanceOf(address(this));
57
58 uint256 sharesMinted = vault.deposit(_amount, _apeVault);
59
60 uint256 afterBal = IERC20(_token).balanceOf(address(this));
61 deposited = beforeBal - afterBal;
62
63
64 ApeVaultWrapperImplementation(_apeVault).addFunds(deposited);
65 emit DepositInVault(_apeVault, _token, sharesMinted);
66 }
```

Listing 2.9: ApeRouter.sol

Impact NA.

Suggestion Use the actual amount for approval instead of the unlimited one.

2.3.2 Use safeTransfer

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description

The `_tapBase` use the `transfer` function while `_tapOnlyProfit` uses `safeTransfer`. It's a good practice to make the invocation consistent and use `safeTransfer` instead.

```
200
201 function _tapOnlyProfit(uint256 _tapValue, address _recipient) internal {
202     uint256 fee = FeeRegistry(ApeRegistry(apeRegistry).feeRegistry()).getVariableFee(_tapValue,
        _tapValue);
203     uint256 finalTapValue = _tapValue + _tapValue * fee / TOTAL_SHARES;
204     require(_shareValue(finalTapValue) <= profit(), "Not enough profit to cover epoch");
205     vault.safeTransfer(_recipient, _tapValue);
206     vault.safeTransfer(ApeRegistry(apeRegistry).treasury(), _tapValue * fee / TOTAL_SHARES);
207 }
208
209 /**
210  * @notice
211  * Used to take funds from vault by deducting a part from profits
212  * @param _tapValue Amount of funds to take
213  * @param _recipient recipient of funds (always distributor)
214  */
215 function _tapBase(uint256 _tapValue, address _recipient) internal {
216     uint256 underlyingTapValue = _shareValue(_tapValue);
217     uint256 profit_ = profit();
218     uint256 fee = FeeRegistry(ApeRegistry(apeRegistry).feeRegistry()).getVariableFee(profit_,
        underlyingTapValue);
219     uint256 finalTapValue = underlyingTapValue + underlyingTapValue * fee / TOTAL_SHARES;
```

```

220     if (finalTapValue > profit_)
221         underlyingValue -= finalTapValue - profit_;
222     vault.transfer(_recipient, _tapValue);
223     vault.transfer(ApeRegistry(apeRegistry).treasury(), _tapValue * fee / TOTAL_SHARES);
224 }

```

Listing 2.10: wrapper/beacon/ApeVault.sol

```

126     receive() external payable {
127         emit Deposit(msg.sender, msg.value, address(this).balance);
128     }
129
130     fallback() external payable {
131         if (msg.value > 0) {
132             emit Deposit(msg.sender, msg.value, address(this).balance);
133         }
134     }

```

Listing 2.11: MultiSigs.sol

Impact NA

Suggestion Use `safeTransfer`.

2.3.3 Ensure that the same storage layout of the proxy

Status Acknowledged.

Description

In our local simulation, proxy behaves unexpectedly when the new implementation of `ApeVault` has a different storage layout. When updating the implementation of the `ApeVault`, ensure that the storage layout is same.

Impact Unexpected behavior.

Suggestion Keep the same storage layout.

Feedback from the developer When we will be pushing out new implementations, it will be mandatory to keep storage layout the way it is and add new storage at the end of the current layout.

2.3.4 Ensure that state variables in ApeRegistry must be properly initialized

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

The state variables (e.g., `treasury`) are not initialized in the `constructor()`, thus the corresponding setters (e.g., `setTreasury()`) must be properly invoked. Otherwise, it will lead to a fund loss since part of the fund will be transferred to the treasury when performing the tapping operation.

```

13     constructor(uint256 _minDelay) TimeLock(_minDelay) {}

```

Listing 2.12: ApeRegistry.sol

Impact NA.

Suggestion Add state variables to the constructor.

2.3.5 Remove unused variables in ApeVault.sol

Status Fixed in [Version 2](#)

Introduced by [Version 1](#)

Description

There are two unused variables in [ApeVault.sol](#), including [hasAccess](#) and [allowanceModule](#).

Impact NA

Suggestion Remove unused variables.