SMART CONTRACT AUDIT REPORT

for

UpDeFi

Prepared By: Yiqun Chen

Hangzhou, China

January 28, 2022

Document Properties

Client	UpDeFi
Title	Smart Contract Audit Report
Target	UpDeFi
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 28, 2022	Shulin Bie	Final Release
1.0-rc	January 25, 2022	Shulin Bie	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
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 UpDeFi, 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 UpDeFi

UpDeFi is a yield farming aggregator running on Binance Smart Chain (BSC), with the goal of optimising DeFi users' yield farming at the lowest possible cost. The protocol has a number of built-in farming strategies and supports multiple farming pools (e.g., PancakeSwap, MarsSwap, etc). The protocol also has its utility token UP, which is distributed to protocol users according to their engagement or contribution.

Table 1.1: Basic Information of UpDeFi

Item	Description
Target	UpDeFi
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 28, 2022

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

- https://github.com/up-defi/UpFarm-hardhat (d878a44)

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

- https://github.com/up-defi/UpFarm-hardhat (2adc051)

1.2 About PeckShield

PeckShield Inc. [11] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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 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

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

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) [9], 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 `UpDeFi` 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	4	■■■■
Low	3	■■■
Informational	0	
Undetermined	0	
Total	7	

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 4 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key UpDeFi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Costly *share* From Improper Deposit Initialization In Strategy	Time and State	Fixed
PVE-002	Low	Timely massUpdatePools In UpFarm::updateUPPerBlock()	Business Logic	Fixed
PVE-003	Low	Potential Sandwich/MEV Attack For _safeSwap()	Time and State	Confirmed
PVE-004	Medium	Suggested Forbidden Transfer For StakingRewards Token	Business Logic	Fixed
PVE-005	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-007	Medium	Minimum Delay Bypass in TimelockController	Time and State	Fixed

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 Possible Costly *share* From Improper Deposit Initialization In Strategy

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `UpFarm/Strategy`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [2]

Description

By design, the `UpFarm` contract is one of the main entries for interaction with users. It organizes a number of farming pools into which supported assets can be staked and implements an incentive mechanism that rewards the staking of different farming pools with the `UP` token. The `Strategy` contract implements the standard farming strategy, while the `StrategyPCS` and `StrategyMars` contracts inheriting from it implement the specific farming strategies.

In particular, one routine, i.e., `Strategy::deposit()`, is called inside the `UpFarm::deposit()` routine when the user stakes his assets, and its returned value representing the pool shares is internally recorded in the `user.shares` in the `UpFarm` contract. While examining the share calculation with the given stakes in the `Strategy::deposit()` routine, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later stakers. Specifically, the issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function deposit( uint256 _wantAmt)
        external
        virtual
        onlyOwner
        nonReentrant
        whenNotPaused
        returns (uint256)
```

```
100     {
101         uint256 beforeAmount =  IERC20(wantAddress).balanceOf(address(this));
102         IERC20(wantAddress).safeTransferFrom(
103             address(msg.sender),
104             address(this),
105             _wantAmt
106         );
107         uint256 afterAmount =  IERC20(wantAddress).balanceOf(address(this));

109         uint256 realamount = afterAmount.sub(beforeAmount);

111         uint256 sharesAdded = realamount;
112         if (wantLockedTotal > 0 && sharesTotal > 0) {
113             sharesAdded = realamount
114                 .mul(sharesTotal)
115                 .mul(entranceFeeFactor)
116                 .div(wantLockedTotal)
117                 .div(entranceFeeFactorMax);
118         }
119         sharesTotal = sharesTotal.add(sharesAdded);

121         if (isAutoComp) {
122             _farm();
123         } else {
124             wantLockedTotal = wantLockedTotal.add(realamount);
125         }
126         emit Deposit(msg.sender,realamount);

128         return sharesAdded;
129     }

131     function farm() external virtual nonReentrant {
132         _farm();
133     }

135     function _farm() internal virtual {
136         require(isAutoComp, "!isAutoComp");
137         uint256 wantAmt = IERC20(wantAddress).balanceOf(address(this));
138         wantLockedTotal = wantLockedTotal.add(wantAmt);
139         IERC20(wantAddress).safeIncreaseAllowance(farmContractAddress, wantAmt);

141         if (isCAKEStaking) {
142             IPancakeswapFarm(farmContractAddress).enterStaking(wantAmt);
143         } else {
144             IPancakeswapFarm(farmContractAddress).deposit(pid, wantAmt);
145         }
146     }
```

Listing 3.1: `Strategy::deposit()&&farm()`

Specifically, when the pool is being initialized, the share value directly takes the value of `realamount` (line 111), which is under control by the malicious actor. As this is the first stake, the current total supply equals the calculated `uint256 sharesAdded = realamount = 1WEI`. With that, the actor can further transfer a huge amount of `wantAddress` tokens to `Strategy` contract with the goal of making per share extremely expensive.

An extremely expensive share can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed share value for staked assets. If truncated to be zero, the staked assets are essentially considered dust and kept by the pool with returning `0` that is internally recorded in the `user.shares` as stake credits.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `Strategy::deposit()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first stake to avoid being manipulated.

Status The issue has been addressed by the following commit: `909c967`.

3.2 Timely massUpdatePools In UpFarm::updateUPPerBlock()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UpFarm`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `UpFarm` contract implements an incentive mechanism that rewards the staking of supported assets with the `UP` token. The rewards are carried out by designating a number of staking pools. The staking users are rewarded in proportional to their staking assets in the pool.

The reward rate (per block) of the `UP` token can be adjusted via the `updateUPPerBlock()` routine. When analyzing its logic, we notice the lack of timely invoking `massUpdatePools()` to update the `accUPPerShare` and `lastRewardBlock` variables before the new reward-related configuration becomes effective. If the call to `massUpdatePools()` is not immediately invoked before updating the reward rate, certain situations may be crafted to create an unfair reward distribution.

```
    function updateUPPerBlock(uint256 _upPerBlock) external onlyOwner {
        UPPerBlock = _upPerBlock;
```

```
        emit UpdateUPPerBlock(msg.sender,_upPerBlock);
    }
```

Listing 3.2: `UpFarm::updateUPPerBlock()`

Recommendation Timely invoke `massUpdatePools()` in the `updateUPPerBlock()` routine.

Status The issue has been addressed by the following commit: `fbed970`.

3.3 Potential Sandwich/MEV Attack For _safeSwap()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StrategyMars/StrategyPCS/Strategy`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

While examining the `Strategy` contract, we notice there is a routine (i.e., `_safeSwap()`) that can be improved with effective slippage control. To elaborate, we show below the related code snippet of the `Strategy` contract. According to the design, the `_safeSwap()` function is used to swap a certain token (specified by the input `_path`) to another one. In the function, the `swapExactTokensForTokensSupporting-FeeOnTransferTokens()` function of `PancakeSwap/MarsSwap` is called (lines 515 - 522) to swap the exact token to another one. However, we observe the second input `amountOutMin` parameter is calculated according to the current state of the `PancakeSwap/MarsSwap` pool (line 511), which may have been price-manipulated. In other words, the slippage control is ineffective and is therefore vulnerable to possible front-running attacks.

```
503     function _safeSwap(
504         address _uniRouterAddress,
505         uint256 _amountIn,
506         uint256 _slippageFactor,
507         address[] memory _path,
508         address _to,
509         uint256 _deadline
510     ) internal virtual {
511         uint256[] memory amounts =
512             IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn, _path);
513         uint256 amountOut = amounts[amounts.length.sub(1)];

515         IPancakeRouter02(_uniRouterAddress)
516             .swapExactTokensForTokensSupportingFeeOnTransferTokens(
517             _amountIn,
```

```
            amountOut.mul(_slippageFactor).div(1000),
            _path,
            _to,
            _deadline
        );
    }
```

Listing 3.3: `Strategy::_safeSwap()`

Note other routines, i.e, `StrategyMars::earn()` and `Strategy::earn()`, that use `addLiquidity()` also lack necessary slippage control.

Recommendation Improve the above-mentioned routine by adding effective slippage control.

Status The issue has been confirmed by the team. The team decides to leave it as is after the risk assessment and intends to use oracle to add effective slippage control in the future version.

3.4 Suggested Forbidden Transfer For StakingRewards Token

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StakingRewards`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

By design, the `StakingRewards` contract provides an incentive mechanism that rewards the staking of supported assets with the `UP` token. Specially, when the user deposits the `UP` token with the calling of `deposit()`, the `LP` token (i.e., `"UP Farms Seed Token"`) will be minted to the user to represent the pool shares. Additionally, the `StakingRewards` contract supports all the standard `ERC20` interfaces (including `transfer()/transferFrom()`) since it inherits from the standard `ERC20` contract. In other words, the `LP` token can be transferred like the standard `ERC20` token. Based on this, we notice there is a potential vulnerability that may result in the withdrawal failure.

To elaborate, we show below the related code snippet of the `StakingRewards` contract. In the `deposit()` function, the following statement is executed to record the user's deposit amount: `user.amount = user.amount.add(realAmount)`, and at the same time the same amount of `LP` token will be minted. In the `withdraw()` function, the `user.amount` will be subtracted from the withdrawal amount of the token and the same withdrawal amount of `LP` token will be burned. This is reasonable under the assumption that the vault's internal asset balances (i.e., `user.amount`) are always consistent with actual token balances maintained in individual ERC20 token contracts. However, we notice the `transfer()` interface of the `StakingRewards` contract is inherited from the standard `ERC20` contract, which only maintains the LP token balances. If we assume Alice transfers the LP token to Bob, both Alice and Bob cannot withdraw the deposit UP token because of the inconsistency between the internal asset records (i.e., `user.amount`) and LP token balances maintained in ERC20 token contracts. We suggest to override the `_transfer()` interface to forbid LP token transfer.

```
    function deposit(uint256 _pid, uint256 _amount) external override validatePid(_pid)
        nonReentrantwhenNotPaused {
        ...
        if (_amount > 0) {
            uint256 beforeAmount =  pool.lpToken.balanceOf(address(this));
            pool.lpToken.safeTransferFrom(
                address(msg.sender),
                address(this),
                _amount
            );
            uint256 afterAmount =  pool.lpToken.balanceOf(address(this));
            uint256 realAmount = afterAmount.sub(beforeAmount);
            if (address(uptoken) == address(pool.lpToken)) {
                _mint(msg.sender, realAmount);
            }
            user.amount = user.amount.add(realAmount);
        }
        user.rewardDebt = user.amount.mul(pool.accUPPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from StakingReward.
    function withdraw(uint256 _pid, uint256 _amount) external override validatePid(_pid)
         nonReentrant {
        ...
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            if (address(uptoken) == address(pool.lpToken)) {
                _burn(msg.sender, _amount);
            }
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accUPPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.4: `StakingRewards::deposit()&&withdraw()`

Recommendation Suggest to override the `_transfer()` interface as above-mentioned.

Status The issue has been addressed by the following commit: `fbed970`.

3.5 Duplicate Pool Detection and Prevention

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UpFarm
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The UpFarm contract provides an incentive mechanism that rewards the staking of supported assets with the UP token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*multiplier/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure or moderated by a privileged account). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in add(), whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool with a duplicate token from being added. Though it is a privileged interface (protected with the modifier onlyOwner), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function add(
        uint256 _allocPoint,
        IERC20 _want,
        address _strat
    ) external onlyOwner {
        massUpdatePools();
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                want: _want,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accUPPerShare: 0,
                strat: _strat
            })
        );
        emit Add(_allocPoint,address(_want),_strat);
```

```
    }
```

Listing 3.5: `UpFarm::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

Status The issue has been addressed by the following commit: `fbed970`.

3.6 Trust Issue Of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `UpDeFi` protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., mint `UpToken` infinitely). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function mint(address _to, uint256 _amount) external {
        require(hasRole(MINTER_ROLE, msg.sender), "Caller is not a minter");
        require(totalSupply().add(_amount) <= cap(), "Out Of the Cap");
        _mint(_to, _amount);
    }

    function increaseCap(uint256 _increaseNum) external {
        require(hasRole(GOVERNOR_ROLE, msg.sender), "Caller is not a Governor");
        _cap = _cap.add(_increaseNum);

        emit IncreaseCap(msg.sender,_increaseNum);
    }
```

Listing 3.6: `UPToken::mint()&&increaseCap()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `UpDeFi` design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed 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 confirmed by the team. The team intends to introduce `multi-sig` and `timelock` mechanisms to mitigate this issue when the protocol is deployed on the mainnet. Additionally, `DAO` governance will be applied once the governance tokens are sufficiently distributed to the public.

3.7 Minimum Delay Bypass In TimelockController

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `TimelockController`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

The `TimelockController`, introduced in `OpenZeppelin Contracts 3.3`, is a smart contract that enforces a delay on all actions directed towards an owned contract. A typical setup is to position the `TimelockController` as the admin of an application smart contract so, whenever a privileged action is to be executed, it has to wait for a certain time specified by the `TimelockController`.

The security benefits of the `TimelockController` are twofold. Firstly, it provides an extra layer of security to a project's team by giving a heads up on every privileged action anticipated in the system. This allows the team to detect and react to malicious calls by compromised admin accounts. Secondly, it protects the community from the project's governance itself, allowing members to exit the protocol if they disagree with any impending changes.

In particular, `scheduleBatch()` (a schedule function) and `executeBatch()` (an execute function), allow the caller to enqueue and execute proposals that run multiple calls in sequence. However, there is a vulnerability in their implementation that can be exploited by the malicious `EXECUTOR` to execute arbitrary tasks bypassing the minimum delay protection.

To elaborate, we show below the related code snippet of the `TimelockController` contract. A malicious `EXECUTOR` could execute a batch with the calling of `executeBatch()`, including a set of calls, i.e., the call to the `TimelockController` itself to clear the minimum delay and grant `PROPOSER` and `ADMIN` rights to an address under their control, the call to `scheduleBatch()` to enqueue the batch by the controlled `PROPOSER` and the call to the arbitrary privileged functions under the `TimelockController` control. By doing so, the malicious `EXECUTOR` effectively takes full control of the `TimelockController` contract.

```
    function executeBatch(
        address[] calldata targets,
        uint256[] calldata values,
        bytes[] calldata datas,
        bytes32 predecessor,
        bytes32 salt
    ) external payable virtual onlyRole(EXECUTOR_ROLE) {
        require(
            targets.length == values.length,
            "TimelockController: length mismatch"
        );
        require(
            targets.length == datas.length,
            "TimelockController: length mismatch"
        );

        bytes32 id =
            hashOperationBatch(targets, values, datas, predecessor, salt);
        _beforeCall(predecessor);
        for (uint256 i = 0; i < targets.length; ++i) {
            _call(id, i, targets[i], values[i], datas[i]);
        }
        _afterCall(id);
    }
```

Listing 3.7: `TimelockController::executeBatch()`

```
    function scheduleBatch(
        address[] calldata targets,
        uint256[] calldata values,
        bytes[] calldata datas,
        bytes32 predecessor,
        bytes32 salt,
        uint256 delay
    ) external virtual onlyRole(PROPOSER_ROLE) {
        require(
            targets.length == values.length,
            "TimelockController: length mismatch"
        );
        require(
            targets.length == datas.length,
            "TimelockController: length mismatch"
        );

        bytes32 id =
            hashOperationBatch(targets, values, datas, predecessor, salt);
        _schedule(id, delay);
        for (uint256 i = 0; i < targets.length; ++i) {
            emit CallScheduled(
                id,
                i,
                targets[i],
                values[i],
```

```
                datas[i],
                predecessor,
                delay
            );
        }
    }
```

Listing 3.8: `TimelockController::scheduleBatch()`

Recommendation Considering the `OpenZeppelin` team has solved this vulnerability, we suggest to upgrade the `TimelockController` to the new version.

Status The issue has been addressed by the following commit: `e698186`.

4 Conclusion

In this audit, we have analyzed the `UpDeFi` design and implementation. `UpDeFi` is a yield farming aggregator running on `Binance Smart Chain` (`BSC`), with the goal of optimising DeFi users' yield farming at the lowest possible cost, which provides a number of built-in farming strategies and supports multiple farming pools (e.g., `PancakeSwap`, `MarsSwap`, etc). The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.