Code Security Assessment

UpDeFi

Mar 2nd, 2022

Table of Contents

Summary

Overview

Project Summary

Audit Summary

Vulnerability Summary

Audit Scope

Findings

GLOBAL-01 : Potential Sandwich Attacks

CKP-01 : Centralization Related Risks In Strategies

CKP-02 : Centralization Related Risks In Farming and Reward Distributions

CKP-03 : Centralization Related Risks In Token And Vesting

CKP-04 : Users Can Never Claim Missed Rewards

CKP-05 : Non-Guaranteed Token Flow

SCK-01 : Flashloan Attack Can Steal Rewards

SCK-02 : Incompatibility With Deflationary Tokens

SCK-03 : Lack of Event Emissions for Significant Transactions

SCK-04 : Function Visibility Optimization

SCK-05 : Rewards When `isCollect` Is `true`

SMC-01 : Redundant Code

SMC-02 : Limits on Fees

SMC-03 : Function Visibility Optimization

SPC-01 : Limits on Fees

SRC-01 : Potential Loss of Pool Rewards

SRC-02 : Incompatibility With Deflationary Tokens

SRC-03 : Redundant Code

SRC-04 : Misspelled Error Message

SRC-05 : Potential Incorrect Return

SRC-06 : Lack of Event Emissions for Significant Transactions

SRC-07 : Function Visibility Optimization

TCC-01 : Usage of `transfer()`

TCC-02 : Potential Reentrancy Attack

TCC-03 : Unused Event

TCC-04 : Lack of Event Emissions for Significant Transactions

TCC-05 : Function Visibility Optimization

TCC-06 : Improper Usage of `tx.origin` for Authorization
UFC-01 : Potential Loss of Pool Rewards
UFC-02 : Incompatibility With Deflationary Tokens
UFC-03 : Redundant Code
UFC-04 : Potential Incorrect Return
UFC-05 : Lack of Event Emissions for Significant Transactions
UFC-06 : Function Visibility Optimization
UFC-07 : Pools With the Same `want` and `strat`
UTC-01 : Lack of Event Emissions for Significant Transactions
UTC-02 : Function Visibility Optimization
VMC-01 : Size of `lockedPeriodAmount` Can Cause Expensive Transactions
VMC-02 : Function Visibility Optimization

Appendix

Disclaimer

About

Summary

This report has been prepared for UpDeFi to discover issues and vulnerabilities in the source code of the UpDeFi project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	UpDeFi
Platform	BSC
Language	Solidity
Codebase	https://github.com/up-defi/UpFarm-hardhat
Commit	• db60cecd25bf17ac90941d20cbff23d7553a3184 • d878a44775610e5afc2293f6849a02d470e41a70 • bcf7234b54b453859fecce5d519ef8a91cb412f4 • 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3 • 99fc180527fe6a775250a5113e8f7e7776546620 • bf29ddf3cd8817a651fb5470ad2145fa8cc7e607

Audit Summary

Delivery Date	Mar 02, 2022
Audit Methodology	Static Analysis, Manual Review

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Mitigated	Resolved
Critical	0	0	0	0	0	0	0
Major	4	0	0	4	0	0	0
Medium	3	0	0	1	0	0	2
Minor	10	0	0	3	0	0	7
Informational	22	0	0	0	0	0	22
Discussion	0	0	0	0	0	0	0

Audit Scope

ID	File	SHA256 Checksum
SRC	StakingRewards.sol	3b5263de2da59e97013625ad9f7b3b215ec30c8628f13366115d9d238cc35e38
SCK	Strategy.sol	52836e05d50cd413a24f8c6e13d634083c4b3e6b30aebec4c1394974db490f2f
SMC	StrategyMars.sol	b36976b4000767c3b79f5457a16bc5d309681d19580642080852e6f0be4229ff
SPC	StrategyPCS.sol	5ce1fdf7f4ed489ae05672c0474e3a1ed390508fecbd506330abed82b3e3af0c
TCC	TimelockController.sol	79b590c45868e8e5ca771ac1cfd38d108eddd3839a44eff2604eaeba67ff3826
UFC	UpFarm.sol	0d34313749161454109 33c04118173074eeedb47fbbffc740fc9b1726227d48d
UTC	UpToken.sol	82bd61d439f703617c8567ddb521a83cac9502280bfc1866dfa33c720671cf5a
VMC	VestingMaster.sol	159fda82cc33362ce322e23b4dde95a6d8b68a7bf5e0b35187a35e359aa0ed55

Review Notes

Overview

UpDeFi is a yield aggregator where users can stake in to earn `UP` tokens as a reward. The collection of contracts looked at contains the staking implementation, the `UP` token, a mechanism that timelocks parts of a user's reward, and their methods for obtaining earnings through PancakeSwap and the Mars Ecosystem.

Dependencies

There are a few depending injection contracts or addresses in the current project:

- `vestingMaster`, a pool's `lpToken`, and `uptoken` for the contract `StakingRewards`;
- `UP`, `vestingMaster`, and a pool's `want` and `strat` for the contract `UpFarm`;
- `vestingToken` for the contract `VestingMaster`;
- `wantAddress`, `token0Address`, `token1Address`, `earnedAddress`, `uniRouterAddress`, `wbnbAddress`, `UPAddress`, `govAddress`, and the addresses in `earnedToUpPath`, `earnedToToken0Path`, `earnedToToken1Path`, `token0ToEarnedPath`, and `token1ToEarnedPath` for the contract `Strategy`;
- `marsRouterAddress`, `UPFarmAddress`, and the addresses in `earnedToWBNBPath` and `wbnbToXmsPath` for the contract `StrategyMars`;
- the addresses in `earnedToWBNBPath` and `UPFarmAddress` for the contract `StrategyPCS`;
- `target`, `targets`, `_autofarmAddress`, `_tokenAddress`, and `_stratAddress` for the contract `TimelockController`.

We assume these contracts or addresses are valid and non-vulnerable actors and implement proper logic to collaborate with the current project.

Privileged Functions

StakingRewards

In the contract `StakingRewards`, the role `GOVERN_ROLE` has the authority over the following functions:

- `StakingRewards.addPool()`, which adds a liquidity pool;
- `StakingRewards.setPool()`, which updates a pool's allocation points;
- `CoreRef.setCore()`, which updates the core address.

In addition, the role `GUARDIAN_ROLE` has the authority over the following functions:

- `StakingRewards.addPool()`, which adds a liquidity pool;

- `StakingRewards.setPool()`, which updates a pool's allocation points.

The role `TIMELOCK_ROLE` has the authority over the following functions:

- `StakingRewards.updateUpPerBlock()`, which decides how many UP tokens are rewarded per block;
- `StakingRewards.updateEndBlock()`, which decides the last block for rewards;
- `StakingRewards.updateVestingMaster()`, which decides the address for `vestingMaster`;
- `CoreRef.pause()`, which prevents the `deposit()` function from being called;
- `CoreRef.unpause()`, which allows the `deposit()` function to be used again.

UpToken

In the contract `UpToken`, the the role `GOVERN_ROLE` has the authority over the following functions:

- `UpToken.mint()`, which mints UP tokens to an address.
- `UpToken.increaseCap()`, which adds to the maximum amount of UP tokens there can be;
- `CoreRef.setCore()`, which updates the core address.

VestingMaster

In the contract `VestingMaster`, the role `GOVERN_ROLE` has authority over the following function:

- `CoreRef.setCore()`, which updates the core address.

In addition, the role `FARM_ROLE` has authority over the following function:

- `VestingMaster.lock()`, which locks up reward amounts for a user.

UpFarm

In the contract `UpFarm`, the role `TIMELOCK_ROLE` has the authority over the following functions:

- `UpFarm.add()`, which adds a liquidity pool;
- `UpFarm.set()`, which changes the allocation points of a pool;
- `UpFarm.setVestingMaster()`, which changes the `vestingMaster` address;
- `UpFarm.updateUPPerBlock()`, which decides the number of UP tokens gained per block;
- `UpFarm.inCaseTokensGetStuck()`, which transfers any non-UP tokens in the contract to the owner;

In addition. the role `GOVERN_ROLE` has the authority over the following function:

- `CoreRef.setCore()`, which updates the core address.

Strategy

In the contract `Strategy`, the role `FARM_ROLE` has the authority over the following functions:

- `Strategy.deposit()`, which increases shares and may stake or deposit to a farm contract;
- `Strategy.withdraw()`, which decreases shares, may unstake or withdraw from a farm ccntract, and send want tokens to the UPFarm address.

In addition, the role `TIMELOCK_ROLE` has the authority over the following functions:

- `Strategy.earn()`, which collects earn tokens, distributes fees, buybacks tokens, and swaps earned tokens for tokens to be added as liquidity to `uniRouterAddress`;
- `Strategy.setSettings()`, which changes the entrance fee, withdraw fee, controller fee, buyback rate, and slippage factor;
- `Strategy.setUniRouterAddress()`, which decides the address of `uniRouterAddress`;
- `Strategy.setBuyBackAddress()`, which decides the address of `buyBackAddress`;
- `Strategy.setRewardsAddress()`, which decides the address of `rewardsAddress`;
- `Strategy.inCaseTokensGetStuck()`, which sends `ERC20` tokens that are not want tokens or earned tokens to an address;
- `Strategy.wrapBNB()`, which wraps the BNB balance of the contract into tokens;
- `CoreRef.pause()`, which prevents the `deposit()`, `earn()`, and `convertDustToEarned()` functions from being called;
- `CoreRef.unpause()`, which allows the functions disabled from `CoreRef.pause()` to be used again.

The role `GOVERN_ROLE` has the authority over the following function:

- `CoreRef.setCore()`, which updates the core address.

TimelockController

In the contract `TimelockController`, the deployer and contract itself has the role `TIMELOCK_ADMIN_ROLE`, which is the administrative role for the roles `TIMELOCK_ADMIN_ROLE`, `PROPOSER_ROLE`, and `EXECUTOR_ROLE`. This means that they have access to the following functions with respect to the aforementioned roles:

- `AccessControl.grantRole()`, which grants one of the above roles to an address;
- `AccessControl.revokeRole()`, which revokes one of the above roles from an address.

The role `PROPOSER_ROLE` has the authority over the following functions:

- `TimelockController.schedule()`, which schedules an operation;
- `TimelockController.scheduleBatch()`, which schedules a batch of operations;
- `TimelockController.cancel()`, which cancels and operation;
- `AccessControl.renounceRole()`, which gives up their role.

The role `EXECUTOR_ROLE` has the authority over the following functions:

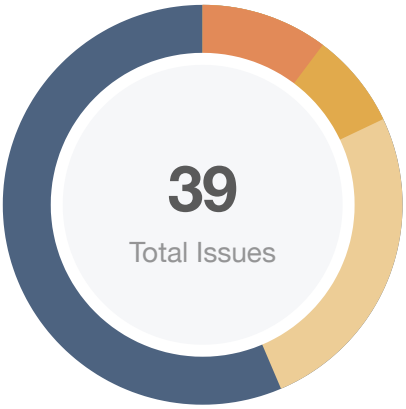
- `TimelockController.execute()`, which executes a scheduled operation;
- `TimelockController.executeBatch()`, which executes a batch of operations;
- `TimelockController.scheduleSet()`, which schedules a change to a liquidity pool's allocation points at an `UpFarm` contract;
- `TimelockController.executeSet()`, which executes a scheduled change to a liquidity pool's allocation points at an `UpFarm` contract;
- `TimelockController.add()`, which adds a liquidity pool to an `UpFarm` contract;
- `TimelockController.earn()`, which causes a `Strategy` contract to collect earned tokens, swap them for liquidity tokens, and then deposit/stake these tokens;
- `TimelockController.farm()`, which causes a `Strategy` contract to deposit/stake tokens;
- `TimelockController.pause()`, which disables the functions `deposit()`, `earn()`, and `convertDustToEarned()` in a `Strategy` contract;
- `TimelockController.unpause()`, which reenables the functions `deposit()`, `earn()`, and `convertDustToEarned()` in a paused `Strategy` contract;
- `TimelockController..wrapBNB()`, which converts a `Strategy` contract's `BNB` balance into `WBNB` tokens.

The contract itself has authority over the following functions, which can be called by having `PROPOSER_ROLE` schedule the operation and `EXECUTOR_ROLE` execute the operation:

- `TimelockController.updateMinDelay()`, which changes the minimum amount of delay for an operation to be executed;
- `TimelockController.updateMinDelayReduced()`, which changes the minimum amount of delay for `set` to be executed at an `UpFarm` contract.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

Findings



Critical	**0** (0.00%)
Major	**4** (10.26%)
Medium	**3** (7.69%)
Minor	**10** (25.64%)
Informational	**22** (56.41%)
Discussion	**0** (0.00%)

ID	Title	Category	Severity	Status
GLOBAL-01	Potential Sandwich Attacks	Logical Issue	Minor	Acknowledged
CKP-01	Centralization Related Risks In Strategies	**Centralization / Privilege**	**Major**	Acknowledged
CKP-02	Centralization Related Risks In Farming and Reward Distributions	**Centralization / Privilege**	**Major**	Acknowledged
CKP-03	Centralization Related Risks In Token And Vesting	**Centralization / Privilege**	**Major**	Acknowledged
CKP-04	Users Can Never Claim Missed Rewards	Logical Issue	Medium	Acknowledged
CKP-05	Non-Guaranteed Token Flow	Logical Issue	Major	Acknowledged
SCK-01	Flashloan Attack Can Steal Rewards	Logical Issue	Medium	Resolved
SCK-02	Incompatibility With Deflationary Tokens	Volatile Code	Minor	Resolved
SCK-03	Lack of Event Emissions for Significant Transactions	Coding Style	Informational	Resolved
SCK-04	Function Visibility Optimization	Gas Optimization	Informational	Resolved
SCK-05	Rewards When `isCollect` Is `true`	Logical Issue	Minor	Acknowledged
SMC-01	Redundant Code	Gas Optimization	Informational	Resolved
SMC-02	Limits on Fees	Logical Issue	Informational	Resolved

ID	Title	Category	Severity	Status
SMC-03	Function Visibility Optimization	Gas Optimization	Informational	Resolved
SPC-01	Limits on Fees	Coding Style	Informational	Resolved
SRC-01	Potential Loss of Pool Rewards	Logical Issue	Minor	Resolved
SRC-02	Incompatibility With Deflationary Tokens	Volatile Code	Minor	Resolved
SRC-03	Redundant Code	Gas Optimization	Informational	Resolved
SRC-04	Misspelled Error Message	Coding Style	Informational	Resolved
SRC-05	Potential Incorrect Return	Volatile Code	Informational	Resolved
SRC-06	Lack of Event Emissions for Significant Transactions	Coding Style	Informational	Resolved
SRC-07	Function Visibility Optimization	Gas Optimization	Informational	Resolved
TCC-01	Usage of `transfer()`	Volatile Code	Minor	Resolved
TCC-02	Potential Reentrancy Attack	Logical Issue	Medium	Resolved
TCC-03	Unused Event	Gas Optimization	Informational	Resolved
TCC-04	Lack of Event Emissions for Significant Transactions	Coding Style	Informational	Resolved
TCC-05	Function Visibility Optimization	Gas Optimization	Informational	Resolved
TCC-06	Improper Usage of `tx.origin` for Authorization	Logical Issue	Minor	Resolved
UFC-01	Potential Loss of Pool Rewards	Logical Issue	Minor	Acknowledged
UFC-02	Incompatibility With Deflationary Tokens	Volatile Code	Minor	Resolved
UFC-03	Redundant Code	Gas Optimization	Informational	Resolved
UFC-04	Potential Incorrect Return	Volatile Code	Informational	Resolved
UFC-05	Lack of Event Emissions for Significant Transactions	Coding Style	Informational	Resolved

ID	Title	Category	Severity	Status
UFC-06	Function Visibility Optimization	Gas Optimization	Informational	Resolved
UFC-07	Pools With the Same `want` and `strat`	Logical Issue	Informational	Resolved
UTC-01	Lack of Event Emissions for Significant Transactions	Gas Optimization	Informational	Resolved
UTC-02	Function Visibility Optimization	Gas Optimization	Informational	Resolved
VMC-01	Size of `lockedPeriodAmount` Can Cause Expensive Transactions	Logical Issue	Minor	Resolved
VMC-02	Function Visibility Optimization	Gas Optimization	Informational	Resolved

GLOBAL-01 | Potential Sandwich Attacks

Category	Severity	Location	Status
Logical Issue	Minor	Global	Acknowledged

Description

The contracts `Strategy`, `StrategyMars` and `StrategyPCS` swap tokens and add liquidity to swapping pools. For example, in `Strategy`:

```
494     function _safeSwap(
495         address _uniRouterAddress,
496         uint256 _amountIn,
497         uint256 _slippageFactor,
498         address[] memory _path,
499         address _to,
500         uint256 _deadline
501     ) internal virtual {
502         uint256[] memory amounts =
503             IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn, _path);
504         uint256 amountOut = amounts[amounts.length.sub(1)];
505 
506         IPancakeRouter02(_uniRouterAddress)
507             .swapExactTokensForTokensSupportingFeeOnTransferTokens(
508             _amountIn,
509             amountOut.mul(_slippageFactor).div(1000),
510             _path,
511             _to,
512             _deadline
513         );
514     }
```

In `StrategyMars`:

```
173                 IPancakeRouter02(marsRouterAddress).addLiquidity(
174                     token0Address,
175                     token1Address,
176                     token0Amt,
177                     token1Amt,
178                     0,
179                     0,
180                     address(this),
181                     block.timestamp.add(600)
182                 );
```

A sandwich attack might happen when an attacker observes a transaction swapping tokens or adding liquidity without setting restrictions on slippage or minimum output amount. The attacker can manipulate the exchange rate by frontrunning (before the transaction is attacked) a transaction to purchase one of the assets and make profits by backrunning (after the transaction is attacked) a transaction to sell the asset.

Although `_slippageFactor` is used in the function `Strategy._safeSwap()`, the amount expected from a swap is calculated through `IPancakeRouter02(_uniRouterAddress).getAmountsOut()`, which depends on the reserves in the swap contract and can be manipulated by flashloans.

An attacker can therefore perform a sandwich attack by manipulating the reserves in the appropriate router.

Recommendation

We recommend using a time-weighted oracle along with the slippage factor to decide how much should be received from a swap.

Alleviation

[**UpDeFi Team**]: This function will be called when compounding the interest for the corresponding strategy. Since the frequency of corresponding calls is at least once a day, the amount of each swap is very small, which is similar to the amount of a common swap transaction on DEX. Therefore, The likelihood and consequences of this sandwich attack are similar to those of attacking a common swap on DEX.

We will also use oracle to add more slippage control later.

[**CertiK**]: The auditors agree that when the amount of each swap is small sandwich attacks are not likely to happen. It would require the `govAddress` role to trigger it regularly.

CKP-01 | Centralization Related Risks In Strategies

Category	Severity	Location	Status
Centralization / Privilege	**Major**	Strategy.sol StrategyMars.sol StrategyPCS.sol	Acknowledged

Description

In the contract `Strategy`, the role `FARM_ROLE` has the authority over the following functions:

- `Strategy.deposit()`, which increases shares and may stake or deposit to a farm contract;
- `Strategy.withdraw()`, which decreases shares, may unstake or withdraw from a farm ccntract, and send want tokens to the UPFarm address.

In addition, the role `TIMELOCK_ROLE` has the authority over the following functions:

- `Strategy.earn()`, which collects earn tokens, distributes fees, buybacks tokens, and swaps earned tokens for tokens to be added as liquidity to `uniRouterAddress`;
- `Strategy.setSettings()`, which changes the entrance fee, withdraw fee, controller fee, buyback rate, and slippage factor;
- `Strategy.setUniRouterAddress()`, which decides the address of `uniRouterAddress`;
- `Strategy.setBuyBackAddress()`, which decides the address of `buyBackAddress`;
- `Strategy.setRewardsAddress()`, which decides the address of `rewardsAddress`;
- `Strategy.inCaseTokensGetStuck()`, which sends `ERC20` tokens that are not want tokens or earned tokens to an address;
- `Strategy.wrapBNB()`, which wraps the BNB balance of the contract into tokens;
- `CoreRef.pause()`, which prevents the `deposit()`, `earn()`, and `convertDustToEarned()` functions from being called;
- `CoreRef.unpause()`, which allows the functions disabled from `CoreRef.pause()` to be used again.

The role `GOVERN_ROLE` has the authority over the following function:

- `CoreRef.setCore()`, which updates the core address.

Any compromise to the aforementioned privileged roles may allow the hacker to take advantage of this and obstruct how the contract should operate.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement;
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles;
  OR
- Remove the risky functionality.

Noted: Recommend considering the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

Alleviation

[**UpDeFi Team**]:

a) Timelock will be applied once the smart contracts are deployed to the mainnet.

b) Multi-sig will be applied once the smart contracts are deployed to the mainnet.

c) DAO governance will be applied once the governance tokens are sufficiently distributed to the public.

CKP-02 | Centralization Related Risks In Farming And Reward Distributions

Category	Severity	Location	Status
Centralization / Privilege	Major	StakingRewards.sol UpFarm.sol TimelockController.sol	Acknowledged

Description

StakingRewards

In the contract `StakingRewards`, the role `GOVERN_ROLE` has the authority over the following functions:

- `StakingRewards.addPool()`, which adds a liquidity pool;
- `StakingRewards.setPool()`, which updates a pool's allocation points;
- `CoreRef.setCore()`, which updates the core address.

In addition, the role `GUARDIAN_ROLE` has the authority over the following functions:

- `StakingRewards.addPool()`, which adds a liquidity pool;
- `StakingRewards.setPool()`, which updates a pool's allocation points.

The role `TIMELOCK_ROLE` has the authority over the following functions:

- `StakingRewards.updateUpPerBlock()`, which decides how many UP tokens are rewarded per block;
- `StakingRewards.updateEndBlock()`, which decides the last block for rewards;
- `StakingRewards.updateVestingMaster()`, which decides the address for `vestingMaster`;
- `CoreRef.pause()`, which prevents the `deposit()` function from being called;
- `CoreRef.unpause()`, which allows the `deposit()` function to be used again.

UpFarm

In the contract `UpFarm`, the role `TIMELOCK_ROLE` has the authority over the following functions:

- `UpFarm.add()`, which adds a liquidity pool;
- `UpFarm.set()`, which changes the allocation points of a pool;
- `UpFarm.setVestingMaster()`, which changes the `vestingMaster` address;

- `UpFarm.updateUPPerBlock()`, which decides the number of UP tokens gained per block;
- `UpFarm.inCaseTokensGetStuck()`, which transfers any non-UP tokens in the contract to the owner;

In addition. the role `GOVERN_ROLE` has the authority over the following function:

- `CoreRef.setCore()`, which updates the core address.

TimelockController

In the contract `TimelockController`, the deployer and contract itself has the role `TIMELOCK_ADMIN_ROLE`, which is the administrative role for the roles `TIMELOCK_ADMIN_ROLE`, `PROPOSER_ROLE`, and `EXECUTOR_ROLE`. This means that they have access to the following functions with respect to the aforementioned roles:

- `AccessControl.grantRole()`, which grants one of the above roles to an address;
- `AccessControl.revokeRole()`, which revokes one of the above roles from an address.

The role `PROPOSER_ROLE` has the authority over the following functions:

- `TimelockController.schedule()`, which schedules an operation;
- `TimelockController.scheduleBatch()`, which schedules a batch of operations;
- `TimelockController.cancel()`, which cancels and operation;
- `AccessControl.renounceRole()`, which gives up their role.

The role `EXECUTOR_ROLE` has the authority over the following functions:

- `TimelockController.execute()`, which executes a scheduled operation;
- `TimelockController.executeBatch()`, which executes a batch of operations;
- `TimelockController.scheduleSet()`, which schedules a change to a liquidity pool's allocation points at an `UpFarm` contract;
- `TimelockController.executeSet()`, which executes a scheduled change to a liquidity pool's allocation points at an `UpFarm` contract;
- `TimelockController.add()`, which adds a liquidity pool to an `UpFarm` contract;
- `TimelockController.earn()`, which causes a `Strategy` contract to collect earned tokens, swap them for liquidity tokens, and then deposit/stake these tokens;
- `TimelockController.farm()`, which causes a `Strategy` contract to deposit/stake tokens;
- `TimelockController.pause()`, which disables the functions `deposit()`, `earn()`, and `convertDustToEarned()` in a `Strategy` contract;
- `TimelockController.unpause()`, which reenables the functions `deposit()`, `earn()`, and `convertDustToEarned()` in a paused `Strategy` contract;

- `TimelockController..wrapBNB()`, which converts a `Strategy` contract's `BNB` balance into `WBNB` tokens.

The contract itself has authority over the following functions, which can be called by having `PROPOSER_ROLE` schedule the operation and `EXECUTOR_ROLE` execute the operation:

- `TimelockController.updateMinDelay()`, which changes the minimum amount of delay for an operation to be executed;
- `TimelockController.updateMinDelayReduced()`, which changes the minimum amount of delay for `set` to be executed at an `UpFarm` contract.

Any compromise to the aforementioned privileged roles may allow the hacker to take advantage of this and obstruct how the contract should operate.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND

- Introduction of a DAO/governance/voting module to increase transparency and user involvement;
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles;
  OR
- Remove the risky functionality.

Noted: Recommend considering the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

Alleviation

[**UpDeFi Team**]:

a) Timelock will be applied once the smart contracts are deployed to the mainnet.

b) Multi-sig will be applied once the smart contracts are deployed to the mainnet.

c) DAO governance will be applied once the governance tokens are sufficiently distributed to the public.

CKP-03 | Centralization Related Risks In Token And Vesting

Category	Severity	Location	Status
Centralization / Privilege	**Major**	UpToken.sol VestingMaster.sol	Acknowledged

Description

UpToken

In the contract `UpToken`, the the role `GOVERN_ROLE` has the authority over the following functions:

- `UpToken.mint()`, which mints UP tokens to an address.
- `UpToken.increaseCap()`, which adds to the maximum amount of UP tokens there can be;
- `CoreRef.setCore()`, which updates the core address.

VestingMaster

In the contract `VestingMaster`, the role `GOVERN_ROLE` has authority over the following function:

- `CoreRef.setCore()`, which updates the core address.

In addition, the role `FARM_ROLE` has authority over the following function:

- `VestingMaster.lock()`, which locks up reward amounts for a user.

Any compromise to the aforementioned privileged roles may allow the hacker to take advantage of this and obstruct how the contract should operate.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement;
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles;
  OR
- Remove the risky functionality.

Noted: Recommend considering the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

Alleviation

[**UpDeFi Team**]:

a) Timelock will be applied once the smart contracts are deployed to the mainnet.

b) Multi-sig will be applied once the smart contracts are deployed to the mainnet.

c) DAO governance will be applied once the governance tokens are sufficiently distributed to the public.

CKP-04 | Users Can Never Claim Missed Rewards

Category	Severity	Location	Status
Logical Issue	Medium	StakingRewards.sol: 345 UpFarm.sol: 297	Acknowledged

Description

When users call `deposit()` or `withdraw()`, they can acquire UP tokens as a reward depending on how much they have deposited and for how long. This transfer of rewards is done through the function `safeUPTransfer()`.

However, if there are not enough UP tokens in the contract, users will receive a smaller reward or even no reward. In these situations, the user's `rewardDebt` is still updated, causing them to never be able to acquire these missed rewards.

Recommendation

We recommend implementing a way for users to claim missed rewards.

Alleviation

[**UpDeFi Team**]: We have developed functions to query the contract's UP tokens equity, which the operations team checks daily. When the UP tokens equity will reach 0 soon, the operations team will transfer UP tokens to the contract to guarantee that the balance in the contract is always greater than the liabilities (user equity).

We've also added some event logs of deposit/withdraw, which can be queried for user missed rewards in the unlikely event that happens. Operations teams can compensate users based on these numbers.

CKP-05 | Non-Guaranteed Token Flow

Category	Severity	Location	Status
Logical Issue	Major	StakingRewards.sol: 371 UpFarm.sol: 313	Acknowledged

Description

The value of `upPerBlock` can be updated through the function `updateUpPerBlock` by the `_owner` role, which decides the number of UP tokens to be distributed to users.

However, it does not transfer tokens to the `StakingRewards` contract if `upPerBlock` is increased. It is possible that the contract does not have enough balance to pay users' pending UP tokens.

Recommendation

We recommend implementing a way to ensure that there are enough tokens in the contract for users' rewards.

Alleviation

[**UpDeFi Team**]: We have developed functions to query the contract's UP tokens equity, which the operations team checks daily. When the UP tokens equity will reach 0 soon, the operations team will transfer UP tokens to the contract to guarantee that the balance in the contract is always greater than the liabilities (user equity).

We've also added some event logs of deposit/withdraw, which can be queried for user missed rewards in the unlikely event that happens. Operations teams can compensate users based on these numbers.

SCK-01 | Flashloan Attack Can Steal Rewards

Category	Severity	Location	Status
Logical Issue	Medium	Strategy.sol: 106	Resolved

Description

If `isAutoComp` is set to `true`, such as in `StrategyMars`, then an attacker can use a flashloan attack to steal rewards.

In the function `deposit()`, a user's shares are calculated before `wantLockedTotal` is changed.

```
104         uint256 sharesAdded = _wantAmt;
105         if (wantLockedTotal > 0 && sharesTotal > 0) {
106             sharesAdded = _wantAmt
107                 .mul(sharesTotal)
108                 .mul(entranceFeeFactor)
109                 .div(wantLockedTotal)
110                 .div(entranceFeeFactorMax);
111         }
112         sharesTotal = sharesTotal.add(sharesAdded);
113 
114         if (isAutoComp) {
115             _farm();
116         } else {
117             wantLockedTotal = wantLockedTotal.add(_wantAmt);
118         }
```

When `isAutoComp` is `true`, the `wantLockedTotal` is instead increased by the balance of the contract.

```
127     function _farm() internal virtual {
128         require(isAutoComp, "!isAutoComp");
129         uint256 wantAmt = IERC20(wantAddress).balanceOf(address(this));
130         wantLockedTotal = wantLockedTotal.add(wantAmt);
```

As such, if `IERC20(wantAddress).balanceOf(address(this))` is higher than `_wantAmt`, then the number of shares created for the user are worth more than they should be.

An attacker can therefore take a flashloan to become a dominant shareholder and withdraw immediately to obtain a large portion of the difference between their deposit and `IERC20(wantAddress).balanceOf(address(this))`.

Recommendation

We recommend updating `wantLockedTotal` before calculating `sharesAdded`.

Alleviation

The UpDeFi team heeded the advice and now calculates shares using the `wantLockedTotal` and the contract's balance prior to a token transfer. This was done in the commits b25e3ec318911c0f783090605fe54e401979625b and bf29ddf3cd8817a651fb5470ad2145fa8cc7e607.

SCK-02 | Incompatibility With Deflationary Tokens

Category	Severity	Location	Status
Volatile Code	Minor	Strategy.sol: 104	Resolved

Description

When transferring standard ERC20 deflationary tokens, the input amount may not be equal to the received amount due to the charged transaction fee. For example, if a user stakes 100 deflationary tokens (with a 10% transaction fee), only 90 tokens actually arrived in the contract. However, the user can still withdraw 100 tokens from the contract, which causes the contract to lose 10 tokens in such a transaction.

Recommendation

We advise the client to regulate the set of pool tokens supported and add necessary mitigation mechanisms to keep track of accurate balances if there is a need to support deflationary tokens.

Alleviation

The UpDeFi team heeded the advice and now checks balances during deposits, allowing the support for deflationary tokens. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

SCK-03 | Lack Of Event Emissions For Significant Transactions

Category	Severity	Location	Status
Coding Style	Informational	Strategy.sol: 90, 159	Resolved

Description

The following functions affects the status of sensitive state variables and should be able to emit events as notifications:

- `deposit()`
- `withdraw()`

Recommendation

Consider adding events for sensitive actions and emit them in the aforementioned functions.

Alleviation

The UpDeFi team heeded our advice and added events `Deposit` and `Withdraw` that are emitted in the functions `deposit()` and `withdraw()`, respectively. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

SCK-04 | Function Visibility Optimization

Category	Severity	Location	Status
Gas Optimization	Informational	Strategy.sol: 90, 123, 204, 334, 379, 383, 387, 435, 440, 445, 454, 463, 472, 490	Resolved

Description

The following functions are declared as `public` and are not invoked in any of the contracts contained within the project's scope:

- `deposit()`
- `farm()`
- `earn()`
- `convertDustToEarned()`
- `pause()`
- `unpause()`
- `setSettings()`
- `setGov()`
- `setOnlyGov()`
- `setUniRouterAddress()`
- `setBuyBackAddress()`
- `setRewardsAddress()`
- `inCaseTokensGetStuck()`
- `wrapBNB()`

The functions that are never called internally within the contract should have external visibility.

Recommendation

We advise that the functions' visibility specifiers are set to `external`, optimizing the gas cost of the functions.

Alleviation

The UpDeFi team heeded our advice and changed the above mentioned functions' visibility specifiers to `external`. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

SCK-05 | Rewards When `isCollect` Is `true`

Category	Severity	Location	Status
Logical Issue	Minor	Strategy.sol: 148	Acknowledged

Description

Users normally earn rewards when `earn()` is called so that earned tokens are swapped into want tokens and staked/deposited. This allows users with shares to claim more than they deposited.

However, when the `isCollect` variable is set to `true`, each time `earn()` is called, the earnings are sent to `rewardsAddress`. Hence when users withdraw their shares, they will never be able to earn these rewards.

Recommendation

We recommend removing this variable or implementing how rewards are distributed from `rewardsAddress`.

Alleviation

[**UpDeFi Team**]: For those strategies where isCollect variable is set to true, the rewards are distributed to users in the following way: the strategies will give users more UP token rewards compared to simple compounding strategies, and the earnings sent to rewardsAddress will be distributed to UP token stakers, so in this case users can not only get the rewards but also get more rewards.

SMC-01 | Redundant Code

Category	Severity	Location	Status
Gas Optimization	Informational	StrategyMars.sol: 66, 93	Resolved

Description

The `require` statements checks if `isAutoComp` is `true`, but this variable is always set to `true` and never changes.

Recommendation

We recommend removing the `require` statements.

Alleviation

The UpDeFi team heeded our advice and removed the redundant code. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

SMC-02 | Limits On Fees

Category	Severity	Location	Status
Logical Issue	Informational	StrategyMars.sol: 58~60	Resolved

Description

The fees set in the constructor of `StrategyMars` can bypass the fee limits defined in `Strategy`.

Recommendation

We recommend having `require` statements in the constructor for the fees so that the same limits in `Strategy` are imposed.

Alleviation

The UpDeFi team heeded the advice and now contains limits on the fees in the constructor. This was done in commit 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3.

SMC-03 | Function Visibility Optimization

Category	Severity	Location	Status
Gas Optimization	Informational	StrategyMars.sol: 92, 222	Resolved

Description

The following functions are declared as `public` and are not invoked in any of the contracts contained within the project's scope:

- `earn()`
- `setMarsRouterAddress()`

The functions that are never called internally within the contract should have external visibility.

Recommendation

We advise that the functions' visibility specifiers are set to `external`, optimizing the gas cost of the functions.

Alleviation

The UpDeFi team heeded our advice and changed the `earn()` function's visibility specifier to `external` while also removing the `setMarsRouterAddress()` function. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

SPC-01 | Limits On Fees

Category	Severity	Location	Status
Coding Style	Informational	StrategyPCS.sol: 51, 55, 56	Resolved

Description

The fees set in the constructor of `StrategyPCS` can bypass the fee limits defined in `Strategy`.

Recommendation

We recommend having `require` statements in the constructor for the fees so that the same limits in `Strategy` are imposed.

Alleviation

The UpDeFi team heeded the advice and now contains limits on the fees in the constructor. This was done in commit 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3.

SRC-01 | Potential Loss Of Pool Rewards

Category	Severity	Location	Status
Logical Issue	Minor	StakingRewards.sol: 99, 128	Resolved

Description

In the `addPool()` and `setPool()` functions, the flag '_withUpdate' determines if all the pools will be updated. This reliance might lead to a significant loss of the reward amount.

```
if (_withUpdate) {
    massUpdatePools();
}
```

For an illustration, assume we only have one pool and it has the values `pool.allocPoint == 50` and `totalAllocPoint == 50`. Now we want to add another pool with `pool.allocPoint == 50`.

There will be two scenarios for calculating the pool reward:

Case 1: `withUpdate` is set to `true`.

- Distribute the reward and update the pool.
- Add the given pool information.

Case 2: `withUpdate` is set to `false`:

- Add the given pool information.

(Note: While we focused on the `addPool()` function, both the `addPool()` and `setPool()` functions update `totalAllocPoint`, which is used in the calculation of pool rewards in the function `updatePool()`

In Case 1, the reward for the first pool is updated in the call to `updatePool()`

```
166         upReward = multiplier.mul(upPerBlock).mul(pool.allocPoint).div(
167             totalAllocPoint
168         );
169     }
```

so we have `upReward = multiplier * upPerBlock * 50 / 50 = multiplier * UPPerBlock`.

In Case 2, an update `totalAllocPoint = totalAllocPoint.add(_allocPoint)` is done first. Then when `updatePool()` is called later, the calculation of the reward for the first pool becomes `upReward = multiplier * upPerBlock * 50 / 100`, half of what it is expected to be.

Recommendation

We advise the client to remove the `_withUpdate` flag and always update pool rewards before updating pool information.

Alleviation

The UpDeFi team heeded the advice and now always updates the pools when `addPool()` or `setPool()` is called. This was done in commit [d878a44775610e5afc2293f6849a02d470e41a70](d878a44775610e5afc2293f6849a02d470e41a70).

SRC-02 | Incompatibility With Deflationary Tokens

Category	Severity	Location	Status
Volatile Code	Minor	StakingRewards.sol: 271	Resolved

Description

When transferring standard ERC20 deflationary tokens, the input amount may not be equal to the received amount due to the charged transaction fee. For example, if a user stakes 100 deflationary tokens (with a 10% transaction fee), only 90 tokens actually arrived in the contract. However, the user can still withdraw 100 tokens from the contract, which causes the contract to lose 10 tokens in such a transaction.

Recommendation

We advise the client to regulate the set of pool tokens supported and add necessary mitigation mechanisms to keep track of accurate balances if there is a need to support deflationary tokens.

Alleviation

The UpDeFi team heeded the advice and now checks balances during deposits, allowing the support for deflationary tokens. This was done in commit 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3.

SRC-03 | Redundant Code

Category	Severity	Location	Status
Gas Optimization	Informational	StakingRewards.sol: 44, 70~71	Resolved

Description

The following areas contain redundant code:

1. (Line 44) The only use of the variable `BONUS_MULTIPLIER` is to multiply an amount in the function `getMultiplier()`. However, the value of `BONUS_MULTIPLIER` is always 1 and never changes, making this multiplication unneeded.
2. (Lines 70-71) The variable `up` is never used except for defining `uptoken`. This variable can be removed and have `uptoken` be defined as `IERC20(_up)`.

Recommendation

We recommend removing these lines of code.

Alleviation

The UpDeFi team heeded the advice and removed the redundant code. This was done in commit [d878a44775610e5afc2293f6849a02d470e41a70](d878a44775610e5afc2293f6849a02d470e41a70).

SRC-04 | Misspelled Error Message

Category	Severity	Location	Status
Coding Style	Informational	StakingRewards.sol: 160	Resolved

Description

In the function `getUPReward()`, the following error message is misspelled:

```
158         require(
159             pool.lastRewardBlock < block.number,
160             "StakingReward::getUPReward: Must little than the current block number"
161         );
```

It should instead read "Must be lower than the current block number."

Recommendation

We recommend fixing the error message.

Alleviation

The UpDeFi team heeded the advice and made the error message clearer. This was done in commit [d878a44775610e5afc2293f6849a02d470e41a70](d878a44775610e5afc2293f6849a02d470e41a70).

SRC-05 | Potential Incorrect Return

Category	Severity	Location	Status
Volatile Code	Informational	StakingRewards.sol: 143	Resolved

Description

The function `getMultiplier()` is correctly called within the contract.

However, considering that it is a `public` function, it should set `_to` to `endBlock` when `_to > endBlock` and `_from` to `startBlock` when `_from < startBlock` so that external users obtain correct information.

Recommendation

It is recommended to modify the implementation of `getMultiplier()` to return the correct value for multiplier.

Alleviation

In commit 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3, the function is now an `internal` function, making this issue obsolete.

SRC-06 | Lack Of Event Emissions For Significant Transactions

Category	Severity	Location	Status
Coding Style	Informational	StakingRewards.sol: 99, 128	Resolved

Description

The following functions affects the status of sensitive state variables and should be able to emit events as notifications:

- `addPool()`
- `setPool()`

Recommendation

Consider adding events for sensitive actions and emit them in the aforementioned functions.

Alleviation

The UpDeFi team heeded the advice and added events `AddPool` and `SetPool` that are emitted in the functions `addPool()` and `setPool()`, respectively. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

SRC-07 | Function Visibility Optimization

Category	Severity	Location	Status
Gas Optimization	Informational	StakingRewards.sol: 99, 128, 229, 278, 326, 366, 375, 394, 402, 406	Resolved

Description

The following functions are declared as `public` and are not invoked in any of the contracts contained within the project's scope:

- `addPool()`
- `setPool()`
- `deposit()`
- `withdraw()`
- `emergencyWithdraw()`
- `updateUpPerBlock()`
- `updateEndBlock()`
- `updateVestingMaster()`
- `setPause()`
- `setUnPause()`

The functions that are never called internally within the contract should have external visibility.

Recommendation

We advise that the functions' visibility specifiers are set to `external`, optimizing the gas cost of the function.

Alleviation

The UpDeFi team heeded the advice and changed the above mentioned functions' visibility specifiers to `external`. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

TCC-01 | Usage Of `transfer()`

Category	Severity	Location	Status
Volatile Code	Minor	TimelockController.sol: 500	Resolved

Description

After EIP-1884 was included in the Istanbul hard fork, it is not recommended to use `.transfer()` or `.send()` for transferring BNB as these functions have a hard-coded value for gas costs making them obsolete as they are forwarding a fixed amount of gas, specifically `2300`. This can cause issues in case the linked statements are meant to be able to transfer funds to other contracts instead of EOAs.

Recommendation

We advise that the linked `.transfer()` and `.send()` calls are substituted with the utilization of the `sendValue()` function from the `Address.sol` implementation of OpenZeppelin either by directly importing the library or copying the linked code.

Alleviation

The concerning code has been removed in commit [2adc05176f4b4336d428bfd74ee0fbb9bd07aee3] [https://github.com/UpDefi-io/UpFarm-hardhat/commit/2adc05176f4b4336d428bfd74ee0fbb9bd07aee3] making this issue obsolete.

TCC-02 | Potential Reentrancy Attack

Category	Severity	Location	Status
Logical Issue	Medium	TimelockController.sol: 306, 470	Resolved

Description

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - `nonReentrant` modifier for the aforementioned functions to prevent reentrancy attack.

Alleviation

The UpDeFi team heeded the advice and the `nonReentrant` modifier for functions at the aforementioned lines. This was done in commit bcf7234b54b453859fecce5d519ef8a91cb412f4.

TCC-03 | Unused Event

Category	Severity	Location	Status
Gas Optimization	Informational	TimelockController.sol: 75	Resolved

Description

There are two `SetScheduled` events and the latter one defined is never used.

Recommendation

We recommend implementing a use case for this event or removing it.

Alleviation

The UpDeFi team heeded the advice and removed this event. This was done in commit [d878a44775610e5afc2293f6849a02d470e41a70](d878a44775610e5afc2293f6849a02d470e41a70).

TCC-04 | Lack Of Event Emissions For Significant Transactions

Category	Severity	Location	Status
Coding Style	Informational	TimelockController.sol: 421, 470, 510, 519, 523, 527, 531, 535	Resolved

Description

The following functions affects the status of sensitive state variables and should be able to emit events as notifications:

- `setDevWalletAddress()`
- `executeSet()`
- `add()`
- `earn()`
- `farm()`
- `pause()`
- `unpause()`
- `wrapBNB()`

Recommendation

Consider adding events for sensitive actions and emit them in the aforementioned functions.

Alleviation

The UpDeFi team heeded our advice and added events that are emitted in the aforementioned functions. This was done in commit 816bd5c318a7439ef5a8374d4dfe50777214d3e4.

TCC-05 | Function Visibility Optimization

Category	Severity	Location	Status
Gas Optimization	Informational	TimelockController.sol: 162, 171, 212, 234, 287, 306, 328, 421, 433, 421, 470, 498, 503, 510, 519, 523, 527, 531, 535	Resolved

Description

The following functions are declared as `public` and are not invoked in any of the contracts contained within the project's scope:

- `getTimestamp()`
- `getMinDelay()`
- `schedule()`
- `scheduleBatch()`
- `cancel()`
- `execute()`
- `executeBatch()`
- `setDevWalletAddress()`
- `scheduleSet()`
- `executeSet()`
- `withdrawBNB()`
- `withdrawBEP20()`
- `add()`
- `earn()`
- `farm()`
- `pause()`
- `unpause()`
- `wrapBNB()`

The functions that are never called internally within the contract should have external visibility.

Recommendation

We advise that the functions' visibility specifiers are set to `external`, optimizing the gas cost of the functions.

Alleviation

The UpDeFi team heeded the advice and changed the above mentioned functions' visibility specifiers to `external`. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

TCC-06 | Improper Usage Of `tx.origin` For Authorization

Category	Severity	Location	Status
Logical Issue	Minor	TimelockController.sol: 426	Resolved

Description

The function `TimelockController.setDevWalletAddress()` checks `tx.origin` before updating `devWalletAddress`:

```
426         require(tx.origin == devWalletAddress, "tx.origin != devWalletAddress");
```

However, this check can be bypassed if `devWalletAddress` interacts with a malicious contract. For example, the attack vector might look like this:

- `devWalletAddress` -> `MaliciousContract` -> `ExecutorContract` -> `TimelockController.execute()` -> `TimelockController.setDevWalletAddress()`

Recommendation

We recommend the UpDeFi team consider if the aforementioned attack vector could happen and do not use `tx.origin` for authorization if it is possible.

Alleviation

The concerning code has been removed in commit [2adc05176f4b4336d428bfd74ee0fbb9bd07aee3] [https://github.com/UpDefi-io/UpFarm-hardhat/commit/2adc05176f4b4336d428bfd74ee0fbb9bd07aee3] making this issue obsolete.

UFC-01 | Potential Loss Of Pool Rewards

Category	Severity	Location	Status
Logical Issue	Minor	UpFarm.sol: 70, 93	Acknowledged

Description

In the `add()` and `set()` functions, the flag '_withUpdate' determines if all the pools will be updated. This reliance might lead to a significant loss of the reward amount.

```
    if (_withUpdate) {
        massUpdatePools();
    }
```

For an illustration, assume we only have one pool and it has the values `pool.allocPoint == 50` and `totalAllocPoint == 50`. Now we want to add another pool with `pool.allocPoint == 50`.

There will be two scenarios for calculating the pool reward:

Case 1: `withUpdate` is set to `true`.

- Distribute the reward and update the pool.
- Add the given pool information.

Case 2: `withUpdate` is set to `false`:

- Add the given pool information.

(Note: While we focused on the `add()` function, both the `add()` and `set()` functions update `totalAllocPoint`, which is used in the calculation of pool rewards in the function `updatePool()`

In Case 1, the reward for the first pool is updated in the call to `updatePool()`

```
163         uint256 UPReward =
164             multiplier.mul(UPPerBlock).mul(pool.allocPoint).div(
165                 totalAllocPoint
166             );
```

so we have `UPReward = multiplier * UPPerBlock * 50 / 50 = multiplier * UPPerBlock`.

In Case 2, an update `totalAllocPoint = totalAllocPoint.add(_allocPoint)` is done first. Then when `updatePool()` is called later, the calculation of the reward for the first pool becomes `UPReward = multiplier * UPPerBlock * 50 / 100`, half of what it is expected to be.

Recommendation

We advise the client to remove the `_withUpdate` flag and always update pool rewards before updating pool information.

Alleviation

[**UpDeFi Team**]: We will timely invoke massUpdatePools() when any pool's weight has been updated.

The reason that we keep the third parameter `_withUpdate` to the set() routine is because many other well-known protocols also keep this parameter for flexibility.

UFC-02 | Incompatibility With Deflationary Tokens

Category	Severity	Location	Status
Volatile Code	Minor	UpFarm.sol: 208	Resolved

Description

When transferring standard ERC20 deflationary tokens, the input amount may not be equal to the received amount due to the charged transaction fee. For example, if a user stakes 100 deflationary tokens (with a 10% transaction fee), only 90 tokens actually arrived in the contract. However, the user can still withdraw 100 tokens from the contract, which causes the contract to lose 10 tokens in such a transaction.

Recommendation

We advise the client to regulate the set of pool tokens supported and add necessary mitigation mechanisms to keep track of accurate balances if there is a need to support deflationary tokens.

Alleviation

The UpDeFi team heeded the advice and now checks balances during deposits, allowing the support for deflationary tokens. This was done in commit 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3.

UFC-03 | Redundant Code

Category	Severity	Location	Status
Gas Optimization	Informational	UpFarm.sol: 136, 222, 223	Resolved

Description

There are instances where `poolInfo[_pid]` is used, but the variable `pool` is already created to point to this structure.

Recommendation

We recommend using the variable `pool` instead of `poolInfo[_pid]` again.

Alleviation

The UpDeFi team heeded the advice and replaced the redundant code. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

UFC-04 | Potential Incorrect Return

Category	Severity	Location	Status
Volatile Code	Informational	UpFarm.sol: 107	Resolved

Description

The function `getMultiplier()` is correctly called within the contract.

However, considering that it is a `public` function, it should set `_from` to `startBlock` when `_from < startBlock` so that external users obtain correct information.

Recommendation

It is recommended to modify the implementation of `getMultiplier()` to return the correct value for multiplier.

Alleviation

In commit 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3, the function is now an `internal` function, making this issue obsolete.

UFC-05 | Lack Of Event Emissions For Significant Transactions

Category	Severity	Location	Status
Coding Style	Informational	UpFarm.sol: 70, 93	Resolved

Description

The following functions affects the status of sensitive state variables and should be able to emit events as notifications:

- `add()`
- `set()`

Recommendation

Consider adding events for sensitive actions and emit them in the aforementioned functions.

Alleviation

The UpDeFi team heeded the advice and added events `Add` and `Set` that are emitted in the functions `add()` and `set()`, respectively. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

UFC-06 | Function Visibility Optimization

Category	Severity	Location	Status
Gas Optimization	Informational	UpFarm.sol: 66, 70, 93, 111, 130, 173, 273, 277, 307, 312, 317	Resolved

Description

The following functions are declared as `public` and are not invoked in any of the contracts contained within the project's scope:

- `poolLength()`
- `add()`
- `set()`
- `pendingUP()`
- `stakedWantTokens()`
- `deposit()`
- `withdrawAll()`
- `emergencyWithdraw()`
- `setVestingMaster()`
- `updateUPPerBlock()`
- `inCaseTokensGetStuck()`

The functions that are never called internally within the contract should have external visibility.

Recommendation

We advise that the functions' visibility specifiers are set to `external`, optimizing the gas cost of the functions.

Alleviation

The UpDeFi team heeded the advice and changed the above mentioned functions' visibility specifiers to `external`. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

UFC-07 | Pools With The Same `want` And `strat`

Category	Severity	Location	Status
Logical Issue	Informational	UpFarm.sol: 84	Resolved

Description

There is no check to see if a previous pool has the same `want` token and `strat`. If two such pools exist with different `allocPoint` values, it does not make sense for users to deposit into the pool with a lower `allocPoint` value.

Recommendation

We recommend having a check to see if a new pool being added already exists.

Alleviation

The UpDeFi team heeded the advice and added a check to see if a strategy has already been used. This was done in commit 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3.

UTC-01 | Lack Of Event Emissions For Significant Transactions

Category	Severity	Location	Status
Gas Optimization	Informational	UpToken.sol: 25	Resolved

Description

The following function affects the status of sensitive state variables and should be able to emit events as notifications:

- `increaseCap()`

Recommendation

Consider adding events for sensitive actions and emit them in the aforementioned functions.

Alleviation

The UpDeFi team heeded the advice and added the event `IncreaseCap`, which is emitted in the function `increaseCap()`. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

UTC-02 | Function Visibility Optimization

Category	Severity	Location	Status
Gas Optimization	Informational	UpToken.sol: 19, 25	Resolved

Description

The following functions are declared as `public` and are not invoked in any of the contracts contained within the project's scope:

- `mint()`
- `increaseCap()`

The functions that are never called internally within the contract should have external visibility.

Recommendation

We advise that the functions' visibility specifiers are set to `external`, optimizing the gas cost of the functions.

Alleviation

The UpDeFi team heeded the advice and changed the above mentioned functions' visibility specifiers to `external`. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

VMC-01 | Size Of `lockedPeriodAmount` Can Cause Expensive Transactions

Category	Severity	Location	Status
Logical Issue	Minor	VestingMaster.sol: 83	Resolved

Description

If the variable `lockedPeriodAmount` is large, not only will the `for` loop in `lock()` be expensive, but a user's `userLockedRewards[user]` record will be very long.

This can cause expensive transactions or reverts if not enough gas is supplied when the functions `lock()`, `claim()`, and `getVestingAmount()` are called.

Recommendation

We recommend placing a limit on how large `lockedPeriodAmount` can be.

Alleviation

The UpDeFi team heeded the advice and added a maximum value of 59 periods for `lockedPeriodAmount`. This was done in commit 2adc05176f4b4336d428bfd74ee0fbb9bd07aee3.

VMC-02 | Function Visibility Optimization

Category	Severity	Location	Status
Gas Optimization	Informational	VestingMaster.sol: 53, 119, 138	Resolved

Description

The following functions are declared as `public` and are not invoked in any of the contracts contained within the project's scope:

- `lock()`
- `claim()`
- `getVestingAmount()`

The functions that are never called internally within the contract should have external visibility.

Recommendation

We advise that the functions' visibility specifiers are set to `external`, optimizing the gas cost of the functions.

Alleviation

The UpDeFi team heeded the advice and changed the above mentioned functions' visibility specifiers to `external`. This was done in commit d878a44775610e5afc2293f6849a02d470e41a70.

Appendix

Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS

AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING

MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

About

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.