AUDIT REPORT

Auditors : Dimitar Dimitrov & Chrisdior

Disclaimer

This report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all the vulnerabilities are fixed - upon the decision of the Customer.

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts.

Document Changelog:

19th December 2023	**Audit Kickoff and Scoping**

Severity classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack
Likelihood - the chance that a particular vulnerability gets discovered and exploited
Severity - the overall criticality of the risk

Security Review Summary

Previously Reviewed commit hash - 2332f79670f67530f71dbdad2f54949893c1fb0d

Audit Scope

The following smart contracts were in scope of the audit:

Precommit hash: 2332f79670f67530f71dbdad2f54949893c1fb0d

- VeUnoDaoYieldDistributor
- VotingEscrow
- SmartWalletChecker

The following number of issues were found, categorized by their severity:

- Critical: 0 issue
- High: 2 issue
- Medium: 2 issues
- Low: 1 issue
- Informational: 5 issues

Summary Table of Our Findings :

ID	TITLE	Severity	Status
[H-01]	Calling notifyRewardAmount will lead to lose of yield for users	High	Fixed
[H-02]	A malicious rewardNotifier can steal all of the users` yield	High	Fixed
[M-01]	Insufficient input validation	Medium	Fixed

[M-02]	Users can be rugged by Governance	Medium	Fixed
[L-01]	SmartWalletChecker can be easily bypassed	Low	Fixed
[I-01]	Pragma statement	Informational	Fixed
[I-02]	NatSpec docs are incomplete	Informational	Fixed
[I-03]	Redundant modifier	Informational	Fixed
[I-04]	Missing event emission	Informational	Fixed
[I-05]	Prefer battle-tested code over reimplementing common patterns	Informational	Fixed

Initial Report Detailed Findings

[H-01] Calling notifyRewardAmount will lead to lose of yield for users

Impact: High, because users` yield can be manipulated

Likelihood: Medium, because a malicious notifier can call it as many times as he wants

Description:

The notifyRewardAmount function takes a yield amount and extends the periodFinish to now + yieldDuration:

periodFinish = block.timestamp + yieldDuration;

If the block.timestamp is less than periodFinish, it will enter the following else block:

```
  ```
 } else {
            uint256 remaining = periodFinish - block.timestamp;
            uint256 leftover = remaining * yieldRate;
```
```

```
            yieldRate = (_amount + leftover) / yieldDuration;
        }
```

It rebases the yield amount and the leftover yields over the yieldDuration period.

This can lead to diluting the yield rate and rewards being dragged out forever by malicious new yield deposits.

Let's take a look at the following example:

1. For the sake of the example, imagine the current yieldRate is 1000 yield / yieldDuration.
2. When 10% of yieldDuration has passed, a malicious notifier calls notifyRewards with _amount = 0.
3. The new yieldRate = 0 + 900 / yieldDuration, which means the yieldRate just dropped by 10%.
4. This can be repeated infinitely. After another 10% of yield time passed, they trigger notifyRewardAmount(0) to reduce it by another 10% again:
   yieldRate = 0 + 720 / yieldDuration.

The yieldRate should never decrease by a notifyRewardAmount call.

Recommendations

There are two potential fixes to this issue:

1. If the periodFinish is not changed at all and not extended on every notifyRewardAmount call. The yieldRate should just increase by yieldRate += amount / (periodFinish - block.timestamp).
2. Keep the yieldRate constant but extend periodFinish time by += amount / yieldRate.

[H-02] A malicious rewardNotifier can steal all of the users` yield

Impact: High, because users` yield can be stolen for no reason

Likelihood: Medium, because a malicious rewardNotifier just has to check which users have approved the contract to spend their tokens

Description:

Calling the notifyRewardAmount function transfers a user input amount of emittedToken (which is the reward token) to address(this):

```

  function notifyRewardAmount(address _user, uint256 _amount) external {
          ...
        // Handle the transfer of emission tokens via `transferFrom` to reduce the number
        // of transactions required and ensure correctness of the emission amount
        emittedToken.safeTransferFrom(_user, address(this), _amount);

```

The problem is that it transfers the tokens from a user-controlled input address. This means that the _amount of reward tokens will be transferred to the VeUnoDaoYieldDistributor contract even though the user did not do this himself. This is especially problematic if the user sets type(uint256).max as the allowance of the contract because in such case all of his emittedToken balance can be drained.

A malicious user who is marked as rewardNotifiers can call this without a problem and transfer tokens from every user who has approved the contract to spend their tokens.

Recommendations

Use msg.sender instead of a user-supplied _user argument, so tokens can only be moved from the caller's account.

[M-01] Insufficient input validation

Impact:
Medium, because a protocol can be broken and the code could give a false calculations

Likelihood:
Medium, as it can be gamed but it needs compromised / malicious owner

Description :

The _yieldDuration param in setYieldDuration() and _newRate0 param in setYieldRate() are missing any constrains and if we have malicious or compromised owner there might be a serious problem.

With the code written this way, the params can be set to 0 or uint256.max which might not be desired at all.

Recommendation

Set reasonable lower and upper constrains for these params.

[M-02] Users can be rugged by Governance

Impact: High as all of the UNO tokens can be transferred to the owner

Likelihood: Low as a malicious/compromised owner is required

Description

Inside VeUnoDaoYieldDistributor, there is a recoverERC20 method intended to recover mistakenly sent tokens to the contract:

```
 ```
  function recoverERC20(
        IERC20 _token,
        uint256 _amount
) external onlyByOwnGov {
        // Only the owner address can receive the recovery withdrawal
        _token.safeTransfer(owner, _amount);
        emit RecoveredERC20(address(_token), _amount);
    }
```
```

However, the owner or the governance can pass any _token and _amount which will be transferred to the owner. This can lead to a scenario where the emittedTokens address is passed and all of the reward tokens are withdrawn from the system, leaving users with nothing.

Recommendations

Consider adding the following check to protect users:

```
```
require(_token != emittedToken, "You are rug-pulling your users!");

```
```

[L-01] SmartWalletChecker can be easily bypassed

The SmartWalletChecker::check function is used to determine if the caller is a smart contract or an EOA. It does so by checking if the extcodesize(account) == 0:

```
 function check(address account) external view returns (bool) {
        uint256 size;
        assembly {
            size := extcodesize(account)
        }
        return size == 0;
    }
}

```

However, this check can be easily bypassed if a smart contract is calling the method within its constructor. During construction time the codesize will be still 0 and the check will pass.

If you want to make sure that an EOA is calling your contract, a simple way is require(msg.sender == tx.origin). However, preventing a contract is an antipattern with security and interoperability considerations.

[I-01] Pragma statement

The pragma statement is different almost everywhere. Some contracts are using floatable pragma. Every version after 0.8.19 will use the PUSH0 opcode, which is still not supported on some EVM-based chains, for example Arbitrum. If the protocol is expected to be deployed on multiple EVM-based chains (Optimism, Arbitrum, Polygon etc), that will be problematic.

Consider using version 0.8.19 so that the same deterministic bytecode can be deployed to all chains.

Even if the protocol is not expected to be deployed on multiple EVM-based chains, always use stable pragma statement to lock the compiler version and to have deterministic compilation to bytecode.

[I-02] NatSpec docs are incomplete

NatSpec documentation is essential for better understanding of the code by developers and auditors and is strongly recommended. Please refer to the NatSpec format and follow the guidelines outlined there.

Example of a good natspec and how it should be in every external function:

```
/**
     * @notice Implements price getting logic. This method is called by
Optimistic Oracle V3 settling an assertion that
     * is configured to use the escalation manager as the oracle. The
interface is constructed to mimic the UMA DVM.
     * @param identifier price identifier being requested.
     * @param time timestamp of the price being requested.
     * @param ancillaryData ancillary data of the price being requested.
     * @return price from the escalation manager to inform the resolution of
the dispute.
     */
    function getPrice(bytes32 identifier, uint256 time, bytes memory
ancillaryData) external returns (int256);

```

[I-03] Redundant modifier

Modifier notYieldCollectionPaused is used only once and that is in getYield() which is in VeUnoDaoYieldDistributor.sol. Because of that, the modifier code can be just pasted in the getYield()'s body.

```
modifier notYieldCollectionPaused() { //@audit used only once
        require(!yieldCollectionPaused, "VeUnoYD: YCP");
        _;
```

```
    }
```

[I-04] Missing event emission

Couple of functions do not emit an event which might not be good for off-chain monitoring.

Emit an event on state change on the following functions in VeUnoDaoYieldDistributor.sol:

- sync()
- toggleGreylist()
- toggleRewardNotifier()
- setPauses()
- setYieldRate()
- setTimelock()

[I-05] Prefer battle-tested code over reimplementing common patterns

Instead of reimplementing your own Ownership contract, use the one provided by OpenZeppelin (Ownable2Step), since it is well-tested and optimized.