OpenZeppelin | security

Origin Arm Audit

ORIGIN

November 5, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-10-15 To 2024-10-21
Languages	Solidity

Total Issues	10 (4 resolved, 2 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	2 (0 resolved, 1 partially resolved)
Notes & Additional Information	7 (3 resolved, 1 partially resolved)

Scope

We audited the [OriginProtocol/arm-oeth] repository, reviewing the code changes introduced in commit [6427782] of pull request [#13].

In scope were the following files:

```
src
└── contracts
    ├── AbstractARM.sol
    ├── LidoARM.sol
    ├── OwnableOperable.sol
    └── Ownable.sol
```

System Overview

The Origin Automated Redemption Manager (ARM) is a product designed to facilitate zero-slippage swapping for redeemable assets, initially focusing on Liquid Staking Tokens (LSTs). ARM pricing is determined by an operator who ostensibly would monitor current market conditions and shrewdly adjust prices. This system is designed to offer swappers better prices when swapping than Automatic Market Makers (AMMs). This would grow the market and drive returns for the liquidity token providers. Market factors that should drive pricing include the current LST withdrawal queue, the prices users could receive in AMMs, and the market's time-value of their tokens.

We audited a specific implementation of the ARM for stETH, named `LidoARM`, which is built on a base contract called `AbstractARM`. The `AbstractARM` contract provides the core functionality for swapping stETH for WETH and vice versa, with prices (`traderate0` and `traderate1`) set by the contract's owner or operator. These prices are nominated in a `PRICE_SCALE` of 36 decimals and can be adjusted by the owner or operator within predefined, sensible limits. For example, it is impossible to set the protocol's sell price for a token to be less than the protocol's buy price for the same token.

The `AbstractARM` also includes a tokenized vault mechanism, where liquidity providers (LPs) can deposit WETH into the vault using the `deposit()` function. In return, liquidity providers receive LP tokens, representing their share of the pool's liquidity. The contract uses the `convertToShares()` and `convertToAssets()` functions to calculate the number of shares issued based on the current available assets in the pool. Liquidity providers can redeem these LP tokens for WETH after a specified claim delay using the `requestRedeem()` and `claimRedeem()` functions.

In the `LidoARM` implementation, additional logic is provided to interface directly with **Lido's withdrawal queue**. The `requestLidoWithdrawals()` function allows the owner or operator of the contract to submit unstaking requests to Lido (i.e. exchanging stETH for ETH). Once the requests are processed, liquidity providers can claim the ETH using the `claimLidoWithdrawals()` function. This ETH is then converted into WETH and returned to the vault for further swapping or redemption by liquidity providers.

Liquidity providers earn profits from the price differential between users swapping stETH for WETH as well as from the difference between the current stETH price and Lido's 1:1 redemption ratio after the unstaking period. The vault can also set a **fee** which is deducted from liquidity providers' profits, and collected by a designated fee collector. Origin has stated their intention for the fee to be collected to the Origin DAO.

Security Model and Trust Assumptions

Code dependencies

The code we audited utilizes v5.0.2 of the OpenZeppelin Contracts and Openzeppelin Contract Upgradeable libraries for casting between number types, creating proxy contracts, and implementing the pool's ERC-20 features. We are familiar with these libraries and expect them to perform without issue.

External Contracts

Within the system, several external contracts have been utilized. The `AbstractARM` contract allows inheriting contracts to have an external withdrawal queue which is factored into pool accounting. For `LidoARM`, this is the lido withdrawal queue which has been documented here. We assume this behaves as described. `AbstractARM` also integrates with a cap manager, a contract that limits the amount of liquidity that can be deposited in the pool. We assume ARM deployers will not abuse this external functionality.

Privileged Roles

There are two key privileged roles within the system: the **owner** and the **operator.** Both have significant influence over the system's functionality, especially in terms of managing Lido withdrawals and setting prices in the pool.

Owner

The owner has the power to set relevant addresses. They can set the fee collector, the cap manager, the operator, as well as pass their ownership on to another address. They can also set market parameters and conduct special market operations including setting the buy and sell prices for stETH, configuring the cross price (which the buy and sell prices must not exceed), and defining the fee charged to liquidity providers.

It is important to note that the owner, alongside the operator, is one of the two roles authorized to request withdrawals from the Lido withdrawal queue. It is assumed that such withdrawal requests will be made in a timely manner.

Operator

The operator has a subset of the owner's privileges. They are responsible for setting the buy and sell prices for stETH in the pool and have the authority to request withdrawals from the Lido withdrawal queue. It is important to note that the operator role is a less trusted role than the owner but should also perform their duties responsibly and promptly.

Market Risks

The liquidity pool acts as a market maker for the liquidity and base assets (WETH and stETH in this case). They bear the risk of either asset appreciating or depreciating while invested in the pool and are compensated for this risk by profiting from the bid-ask spread. Liquidity providers should consider how their participation in this contract fits into their overall investment program and understand the market forces that act on the assets of interest.

The operator is able to set the prices for the liquidity and base assets with some restrictions. If prices are set incorrectly, relative to market conditions, one of the tokens could very quickly be drained from the contract if large arbitrage trades become profitable against the vault. It is important to understand the potential impact of setting prices relative to market prices. Specifically, changing the prices may introduce arbitrage opportunities between the pool and other exchanges, and is expected that MEV bots will quickly seize the arbitrage opportunity. It is critical to understand the size of the arbitrage trades and how this will impact the token balances in the contract. Furthermore, the contract should not rely on MEV bots to take these opportunities as under extreme market stress scenarios, it may not be profitable for them to make the trade.

Medium Severity

M-01 Potential Denial of Service in `setCrossPrice()`

The `setCrossPrice()` function of the `AbstractARM` contract contains a check that can be exploited to block price adjustments. Specifically, the function requires that the contract's balance of the `baseAsset` (stETH) must be less than `MIN_TOTAL_SUPPLY` when lowering the `crossPrice`. An attacker could exploit this by front-running the `setCrossPrice()` call and sending a small amount of stETH to the contract (up to `MIN_TOTAL_SUPPLY`), preventing the cross price from being updated to a more competitive rate. This attack could disrupt the contract's ability to adjust prices in line with market conditions and limit profit opportunities from the bid-ask spread.

Consider adjusting the logic of the smart contract to implement a way to handle unintended deposits of the `baseAsset`. In the case of the `LidoARM`, this could be solved by calling `requestLidoWithdrawals()` in the same transaction. However, we cannot ensure that this approach would work with all LSTs.

Update: *Resolved in pull request #42 at commit 035a85f. The Origin Protocol team stated:*

> *There are several options to prevent this:*
>
> 1. *`setCrossPrice` is changed to call `requestLidoWithdrawals` if the cross price is being lowered. This is a little complicated as `requestLidoWithdrawals` is in `LidoARM` and not `AbstractARM`. We would have to introduce an abstract withdrawal function that `setCrossPrice` calls and is implemented in the LidoARM contract.*
> 2. *The threshold is increased from `MIN_TOTAL_SUPPLY` which is only 1e12. One whole ETH (1e18) is high enough to stop most attacks but is also high enough that the assets per share could go down if there was only a small amount of funds in the ARM.*
> 3. *Send the `setCrossPrice` transaction privately.*
> 4. *Call `requestLidoWithdrawals()` before `setCrossPrice` when the Owner constructs the transaction in the multisig (Safe). The `setCrossPrice` Natspec*

> *already states that "The base assets should be sent to the withdrawal queue before the cross price can be lowered."*

Low Severity

L-01 Abstract Contracts Allow Direct Modification of State Variables

Defining state variables as `internal` or `public` within `abstract` contracts allows them to be directly modified by child contracts. This may break the expected properties for the state variables and limit off-chain monitoring capabilities due to the lack of event emissions for changes to the variables.

In particular, several state variables in the `AbstractARM` contract have `public` visibility which could cause important system invariants to be broken by child contracts. For example, a child contract could set `crossPrice < traderate1`, which would break the invariant required to ensure that the contract cannot be drained of tokens by swapping.

Consider using `private` visibility for state variables in abstract contracts. In addition, consider creating `internal` functions for updating those variables which emit appropriate events, and verifying if the desirable conditions are met.

Update: *Acknowledged, not resolved. The Origin Protocol team stated:*

> *The trade-off for making the state variables private is the accompanying `external view` functions that add more code to the contract. Currently, `LidoARM` is the only contract that implements `AbstractARM` and it does not write to any of `AbstractARM`'s state variables. Given the simple inheritance structure, we will keep the `public` state variables in `AbstractARM`.*

L-02 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified. A non-exhaustive list is given below.

- In `AbstractARM.sol`, the `AbstractARM` abstract contract
- In `AbstractARM.sol`, the events

- In `LidoARM.sol`, the events
- In `Ownable.sol`, the `Ownable` contract
- In `Ownable.sol`, the `AdminChanged` event
- In `OwnableOperable.sol`, the `OwnableOperable` contract
- In `OwnableOperable.sol`, the `OperatorChanged` event

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Partially resolved in pull request #45 at commit ee3ef88. The Origin Protocol team stated:*

> *The standard Origin contract-level NatSpec has been added. While Origin has not traditionally added Natspec for events, we are considering adding it going forward.*

Notes & Additional Information

N-01 Use Custom Errors

Since Solidity version 0.8.4, custom errors provide a cleaner and more cost-efficient way to explain to users why an operation failed.

For conciseness and gas savings, consider replacing `require` and `revert` messages with custom errors in the `AbstractARM` contract.

Update: *Acknowledged, not resolved. The Origin Protocol team stated:*

> *For consistency, Origin will continue using `require` and `revert` messages. New contract repos are likely to use custom errors.*

N-02 General Code Improvements

- In line 147 of `AbstractARM.sol`, the comment contains a typo: **"The deployer that calls initialize has to approve the this ARM's proxy contract to transfer 1e12**

WETH". It should be corrected to: **"The deployer that calls initialize has to approve the ARM's proxy contract to transfer 1e12 WETH"**.

- The name of the `claimable` function in `AbstractARM.sol` suggests that the function returns the amount claimable by LP providers at call time. However, it returns the total amount that has been claimed plus the balance the contract has of the liquidity asset. This could be misleading for users. Consider renaming the `claimable` function to something more descriptive, adding clear notes of this behavior in the code. Alternatively, consider making the function `internal`.

Update: *Partially resolved in pull request #44 at commit 6384755. The Origin Protocol team stated:*

> *The first change has been made. We will leave the second suggestion as the front-end and analytics are already using the `claimable` function.*

N-03 Upgradeable Storage Cohesion

In order to stop upgrades from overwriting storage variables, `Ownable` and the imported `ERC20Upgradeable` use namespaced storage whereas `OwnableOperable` and `AbstractARM` use gap variables. Furthermore, `Ownable` chooses storage slots as per ERC-1967 while `ERC20Upgradeable` determines them using ERC-7201.

To simplify upgrades and minimize the risk of upgrading incorrectly, consider using one storage scheme for all contracts. OpenZeppelin considers EIP-7201 to be the current best practice.

Update: *Acknowledged, not resolved. The Origin Protocol team stated:*

> *For consistency, Origin will continue using gap variables for now.*

N-04 Use `calldata` instead of `memory`

When dealing with the parameters of `external` functions, it is more gas-efficient to read their arguments directly from `calldata` instead of storing them to `memory`. `calldata` is a read-only region of memory that contains the arguments of incoming `external` function calls. This makes using `calldata` as the data location for such parameters cheaper and more efficient compared to `memory`. Thus, using `calldata` in such situations will generally save gas and improve the performance of a smart contract.

Within `LidoARM.sol`, multiple instances where function parameters should use `calldata` instead of `memory` were identified:

- The `amounts` parameter
- The `requestIds` parameter

Consider using `calldata` as the data location for the parameters of `external` functions to optimize gas usage.

Update: *Resolved in pull request #40 at commit a45e527. The Origin Protocol team stated:*

> *The `requestLidoWithdrawals` and `claimLidoWithdrawals` functions in `LidoARM` have been changed to use `calldata`.*

N-05 Disable Initializers

The `LidoARM` contract was written to be an implementation contract for a proxy contract. A best practice with this pattern is to disable as much as possible in the implementation contract to minimize the attack surface area.

Consider calling _disableInitializers() in initializable contract constructors to prevent malicious actors from front-running initialization.

Update: *Resolved in pull request #43. The Origin Protocol team stated:*

> *`_disableInitializers()` has been added to LidoARM's constructor.*

N-06 Boolean Literal Used in Comparison

Within `AbstractARM.claimRedeem`, there is a comparison between a boolean variable and `false` value. This comparison is unnecessary and removing such comparisons makes the code more gas efficient and less error-prone.

Consider changing the aforementioned check to `require(!request.claimed, "Already claimed")`.

Update: *Acknowledged, not resolved. The Origin Protocol team stated:*

> *We think that explicit comparison to `false` is clearer. If `request.claimed` is false, then continue, else throw an error with `Already claimed`.*

> `require(request.claimed == false, "Already claimed");`

N-07 Redundant Casting

In `AbstractARM`'s constructor, `token0` and `token1` are being cast from `address` to `address`.

Consider removing the aforementioned redundant casting.

Update: *Resolved in pull request #41 at commit bd1f5fa. The Origin Protocol team stated:*

> *The cast was needed as it was using the storage variables of type `IERC20`. We have changed the logic to use the parameters which are of type `address`. `baseAsset = _liquidityAsset == _token0 ? _token1 : _token0;`*

Conclusion

We audited Origin's Automatic Redemption Manager (ARM) contracts which enable zero-slippage swapping of redeemable assets by having a market operator setting fixed prices based on current market rates. This particular ARM is a swap market between stETH and WETH. We found the contract to be very well-thought-out, its features well-documented, and its quality to be high. We want to thank Origin for making themselves available and responsive throughout the audit period. We look forward to this market's success on-chain and in DeFi generally.