yAudit

December 19, 2025

Prepared for
Origin

Audited by
Panda
Watermelon

Origin-ARM

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Origin ARM (Automated Redemption Manager) is a suite of ERC4626 vaults designed to provide efficient liquidity management for liquid staking tokens. The protocol supports multiple liquid staking derivatives including Lido's stETH and EtherFi's weETH, acting as a liquidity pool that enables swaps between staking assets and ETH. ARM generates yield by exploiting the conversion rate difference: when users swap ETH for staking assets, they receive the assets at the favorable market rate (e.g., 1:1.01), while redemptions through the underlying protocol's exit queues return ETH at a 1:1 ratio. This mechanism allows ARM to capture the staking yield premium while providing immediate liquidity to users.

1.2 Audit Scope

This audit covers nine smart contracts totaling 916 lines of code across five days of review.

```
src/contracts
├── AbstractARM.sol
├── CapManager.sol
├── EtherFiARM.sol
├── LidoARM.sol
├── Ownable.sol
├── OwnableOperable.sol
├── Proxy.sol
└── markets
    ├── Abstract4626MarketWrapper.sol
    └── MorphoMarket.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	3
Informational	2

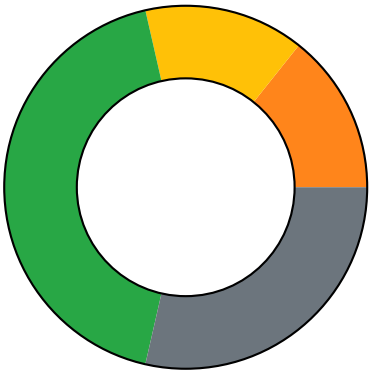

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The Origin ARM protocol demonstrates solid architecture with well-structured ERC4626 implementations. The codebase follows established patterns and includes proper access controls. One high-severity issue was identified regarding fixed conversion rates not accounting for validator slashing, which could lead to unfair loss distribution among users.

2 Audit Overview

2.1 Project Information

Protocol Name: Origin
Repository: https://github.com/OriginProtocol/arm-oeth
Commit Hash: 89be5771adae213636ad205ec297dbf4cd966e1d
Commit URL: https://github.com/OriginProtocol/arm-oeth/blob/89be5771adae213636ad205ec297dbf4cd966e1d/

2.2 Audit Team

Panda, Watermelon

2.3 Audit Timeline

The audit was conducted from November 24 to 28, 2025.

2.4 Audit Resources

Code repositories and documentation Additional resources: whitepaper, previous audits, etc.

Category	Mark	Description
Access Control	Good	The protocol implements proper access control mechanisms using Ownable and OwnableOperable patterns. Role-based permissions are appropriately applied to sensitive functions such as token transfers, market management, and administrative operations. The owner and operator roles are well-defined and consistently enforced throughout the codebase.
Mathematics	Good	The mathematical operations in the protocol are generally sound, utilizing safe arithmetic and proper conversion functions. However, the high-severity finding regarding fixed conversion rates during redemption requests reveals a gap in handling edge cases where underlying asset values decrease due to validator slashing. The protocol correctly implements ERC4626 share-to-asset conversions but lacks dynamic adjustment mechanisms for adverse events.
Complexity	Good	The codebase maintains reasonable complexity. The architecture follows clear separation of concerns with abstract base contracts and specialized implementations for different liquid staking protocols.
Libraries	Excellent	The protocol makes excellent use of battle-tested libraries and standards including OpenZeppelin contracts. Integration with established DeFi protocols like Lido, EtherFi, and Morpho demonstrates proper adherence to external interfaces and standards. No concerning custom implementations of critical functionality were identified.
Decentralization	Good	The protocol relies on centralized owner and operator roles for key operations including market management, token recovery, and withdrawal processing.
Code Stability	Excellent	The codebase demonstrates high stability with mature design patterns and clear structure.
Documentation	Good	The code includes reasonable inline documentation with explanatory comments for key functions and state variables.
Monitoring	Good	The protocol includes standard ERC4626 events for deposits, withdrawals, and transfers.
Testing and verification	Good	The protocol includes a comprehensive testing suite with multiple test types including unit tests, fork tests, and invariant fuzzing. Core contracts demonstrate strong test coverage with thorough edge case validation and property-based testing.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

2.6.1 Fixed conversion rate in withdrawal queue does not account for validator slashing

Technical Details

The `requestRedeem()` function in `AbstractARM.sol` converts shares to assets using the current conversion rate and locks in this amount:

```
function requestRedeem(uint256 shares) external returns (uint256 requestId, uint256
assets) {
    // Calculate the amount of assets to transfer to the redeemer
    assets = convertToAssets(shares);

    // ... store the withdrawal request with fixed assets amount
    withdrawalRequests[requestId] = WithdrawalRequest({
        withdrawer: msg.sender,
        claimed: false,
        claimTimestamp: claimTimestamp,
        assets: SafeCast.toUint128(assets),
        queued: queued
    });
```

The conversion rate is fixed at the time of the request. However, if a significant slashing event occurs (e.g., LIDO validators are slashed) between the `requestRedeem()` call and the `claimRedeem()` call, the actual value of the underlying assets will decrease. The protocol does not adjust the promised withdrawal amount to reflect this loss.
When `claimRedeem()` is executed, it attempts to transfer the originally calculated asset amount:

```
function claimRedeem(uint256 requestId) external returns (uint256 assets) {
    WithdrawalRequest memory request = withdrawalRequests[requestId];

    assets = request.assets; // Fixed amount from request time

    // ... attempts to withdraw this exact amount
    if (assets > liquidityInARM) {
        uint256 liquidityFromMarket = assets - liquidityInARM;
        IERC4626(activeMarketMem).withdraw(liquidityFromMarket, address(this), address(
this));
    }
```

Impact

High.

1. **Unfair Loss Distribution**: Users who have pending withdrawal requests maintain their claim on the original asset amount, while active depositors (those who haven't requested withdrawals) absorb the entire loss from the slashing event. This creates an inequitable distribution of losses.

2. **Bank Run Scenario**: In extreme cases, if a massive slashing event occurs and all users rush to request withdrawals before the conversion rate adjusts, the protocol may become insolvent. Early withdrawal requesters lock in favorable rates while later claimants face insufficient liquidity.

3. **Protocol Insolvency**: The combination of:

- Fixed withdrawal amounts that don't decrease with slashing
- Reduced actual asset backing due to slashing
- Multiple withdrawal requests at pre-slash rates
  Can result in the protocol being unable to honor all withdrawal claims, as the check `require(request.queued <= claimable(), "Queue pending liquidity");` will eventually fail when the total promised withdrawals exceed available liquidity.

Recommendation

Implement a **minimum value mechanism** that compares the conversion rate at request time with that at claim time and uses the lower of the two.
Note: While this solution addresses the core issue, it is not perfect. The timing of when losses materialize (when ETH is actually returned from the liquid staking protocol) means there can still be some variance in how losses are distributed during the claim period. However, this approach significantly improves fairness compared to the current implementation.

Developer Response

Origin Design
The original design of Lido was redeemers would lock in the WETH they would receive when they could claim at the time of their request. Since the redeemers have their ARM LP tokens burned on the redemption request, they do not earn any yield and should also not be penalised for any slashing after their redeem request. The remaining ARM LPs would take the hit from a Lido slashing.
This logic works fine if there is enough liquidity in the ARM or the lending platform to cover the redeem request. It becomes "unfair" if all of the ARM assets are in the Lido withdrawal queue, there is a significant slashing of Lido validators and some ARM LPs request a withdrawal from the ARM before Lido's finalization process to value stETH. The ARM LPs that requested a redeem will get their WETH value before the slashing while the remaining LPs take the hit. If enough ARM LPs see Lido is slashed and get their redeem requests in, there can be no more assets left for later redeemers.
Lido Finalization Process
Lido has a daily rebase process to value the ETH in its validators. This effectively updates the stETH share price of which then increases the stETH balance. This process happens on-chain at around 12.20 pm UTC each day.
In the 2.5 year history of stETH, there has never been a day where the share price was below the previous day. See the following dune dashboard.
For stETH in the Lido Withdrawal Queue, it is valued by Lido as the smaller of

1. The stETH amount at the time of the withdrawal request.
2. The stETH amount at the time the request was finalized.

The second amount has always been the larger amount but will be the smaller amount if there was a large slashing.

The amount received from the Lido withdrawal request does not get any smaller after the request has been finalized.
Under normal Lido ARM operations, the ARM's finalized Lido withdrawal requests are claimed within minutes of the Lido finalization process. That means if there was a large Lido slashing, the ARM's assets per share will be reduced soon after the Lido finalization process if the ARM's requests were finalized. If the ARM's Lido withdrawal requests were not finalized, the ARM will incorrectly value them at a higher value. That is, it will value the outstanding withdrawal requests at the Lido share rate at the time of the withdrawal request and not the now reduced Lido share rate.
A few hundred Lido validators being slashed 1 ETH will not result in a lose to stETH or ARM LPs. Lido earns enough execution and consensus rewards per day to cover this across its 265k validators.

ARM Performance Fees

The ARM calculates its performance fee using a high water mark. This means if the ARM's assets per shares drops, a performance fee will not be collected until the ARM's assets per share goes above the previous high.

What has been changed with this audit

If the ARM's assets per shares has decreased since the redeem was requested, the asset value of the redeemed shares at the time the redeem is claimed is used.
This approach works well if all the ARM's Lido withdrawal requests are finalized each day. If ARM redeemers need liquidity from the Lido withdrawal queue to claim their ARM redemptions, then as soon as the Lido withdrawal requests are claimed, the ARM's assets per share will be reduced if Lido was significantly slashed. The redeemers will then get a reduced amount of WETH when they claim their ARM redeem request. This is fair.
It's not so fair if some of the ARM's Lido withdrawal requests take many days to claim. This is because the outstanding Lido withdrawals will be priced at the Lido share rate at the time of the request and not the now reduced Lido share rate. This means the ARM's assets per share is higher than what it should be. Earlier ARM redeems will get a higher assets per share than later redeemers.
The ARM redeemer could wait until the ARM's assets per share is equal or above the ARM's assets per share at the time of their redeem request. They would then get the WETH calculated at the time of the redeem request. This can be considered unfair if at the ARM redeemer held most of the ARM LP supply at the time of the slashing and new ARM deposits is used to generate yield. For example

- There was only 1 ARM LP and all the ARM assets are in the withdrawal queue
- The ARM LP requests a withdrawal of all their ARM shares
- Lido has a significant slashing event
- The ARM claims the Lido withdrawal receiving a reduced ETH amount
- The ETH is converted to WETH is reserved for the ARM's redeem request
- The ARM receives new deposits
- The new liquidity generates yield which increases the ARM's assets per share
- The new ARM LPs are losing yield to pay for previous loses

A fairer way to socialise Lido loses

If the ARM Operator detects Lido has had a lose, the ARM is paused and all ARM redeem requests are reversed until all Lido withdrawal requests have been claimed. This is the fairest as all ARM LPs, including the redeemers, take a proportional lose and no new LPs are added until after the loses are socialised
The ARM is not currently pausable so that will need to be added. The LP functions `deposit`, `requestRedeem` and `claimRedeem` will need pausing. More Lido withdrawal requests should

be paused. Swaps should also be paused as it increases the amount of stETH owned by the ARM. Claiming Lido withdrawals should be allowed.
The ARM can only be unpaused after all the ARM's Lido withdrawal requests have been claimed. This may take many days if Lido had a lot of withdrawal requests.
ARM LPs with unclaimed redeems will have had their ARM LP tokens burned. These ARM LP tokens would need to be restored so the reduced assets per share can be fairly calculated. A new function would be required to loop through and reverse all outstanding withdrawal requests. This will be complete when the ARM's `withdrawsQueued` amount equals its `withdrawsClaimed` amount.
The ARM should be pausable by the Operator to prevent ARM LPs exiting before the Lido finalization process where the ARM will detect the Lido share rate has dropped, hence Lido has been slashed. The Operator should not have to wait until the Lido finalization process to pause.

2.7 Medium Findings

2.7.1 Asset tokens claimed via merkle distributor or airdropped are permanently locked in Abstract4626MarketWrapper

Technical Details

The `Abstract4626MarketWrapper` contract includes a `merkleClaim()` function that allows claiming tokens from a Merkle Distributor. Additionally, tokens can be accidentally airdropped to the contract. To recover such tokens, the contract provides a `transferTokens()` function. However, the `transferTokens()` function contains an overly restrictive requirement on line 239:

```
1 require(token != asset && token != market, "Cannot transfer asset or market token");
```

This restriction prevents the recovery of any tokens matching the `asset` address (e.g., WETH for Morpho markets, stETH for other markets). While this check is intended to protect deposited assets, it fails to distinguish between: 1. Assets legitimately deposited and represented by market shares 2. Extra asset tokens received through merkle claims or airdrops Since the wrapper contract should only hold market shares under normal operation (not loose asset tokens), any asset tokens in the contract beyond what's needed to maintain the share position represent additional value that cannot be recovered.
The `merkleClaim()` function can claim any token, including the asset token itself, but if the claimed token is the asset token, it becomes permanently locked due to the `transferTokens()` restriction.
Medium. Asset tokens (matching the market's underlying asset) that are claimed via the merkle distributor or accidentally sent/airdropped to the contract will be permanently locked and unrecoverable.

Recommendation

No asset tokens should remain in the wrapper contract. Consider removing the check in `transferTokens` or updating the `deposit()` to use the balance of assets instead of relying on the `assets` amount.

Developer Response

Fixed in PR #167.

2.8 Low Findings

2.8.1 Unreachable defensive code in `claimEtherFiWithdrawals()`

Technical Details

The `claimEtherFiWithdrawals()` function in `EtherFiARM.sol` contains defensive code to handle the edge case where EtherFi withdrawal request NFTs are transferred to the ARM contract from external addresses. However, this code is unreachable due to an earlier validation check.
At lines 122-131, the function validates each withdrawal request:

```
for (uint256 i = 0; i < requestIds.length; i++) {
    // Read the requested amount from storage
    uint256 requestAmount = etherfiWithdrawalRequests[requestIds[i]];

    // Validate the request came from this EtherFi ARM contract and not
    // transferred in from another account.
    require(requestAmount > 0, "EtherFiARM: invalid request");

    totalAmountRequested += requestAmount;
}
```

The `require(requestAmount > 0)` check at line 128 ensures that only withdrawal requests stored in the `etherfiWithdrawalRequests` mapping (i.e., those created by the ARM contract itself via `requestEtherFiWithdrawals()`) can be claimed.
However, later in the function, there's defensive code meant to handle transferred withdrawal requests:

```
if (etherfiWithdrawalQueueAmount < totalAmountRequested) {
    // This can happen if a EtherFi withdrawal request was transferred to the ARM
contract
    etherfiWithdrawalQueueAmount = 0;
} else {
    etherfiWithdrawalQueueAmount -= totalAmountRequested;
}
```

The issue: If a withdrawal request NFT is transferred to the ARM contract from another address, it will not have an entry in the `etherfiWithdrawalRequests` mapping. When attempting to claim it, `requestAmount` will be 0, causing the `require(requestAmount > 0)` check at line 128 to revert with "EtherFiARM: invalid request". Therefore, the defensive code at lines 134-139 can never be reached.

Impact

Low. This is a code clarity issue with no security or functional impact.

Recommendation

The code and comments should be aligned to reflect the actual behavior. There are two possible approaches:
Option 1: Remove the unreachable defensive code and update the comment to clarify that transferred requests cannot be claimed
Option 2: If the intent is to support transferred withdrawal requests, remove the restrictive validation and implement proper handling

Developer Response

Fixed in PR #169

2.8.2 Exact-in swaps do not enforce min-out against actual received amount for under-transferring tokens

Technical Details

The exact-in swap functions compute a theoretical `amountOut` and enforce `amountOut >= amountOutMin`, but do not verify the actual number of output tokens received by the recipient. For tokens like stETH that can transfer up to 2 wei less than requested, users may receive less than their specified `amountOutMin` even though the slippage check passes.

Impact

Low: the slippage check for swaps with stETH as output may be violated by up to 2 wei.

Recommendation

Within `LidoARM`, execute the slippage check against the effective change in the receiver's balance.

Developer Response

We want to keep the swaps as gas efficient as possible so we will not add this check. I think it's just part of using stETH in the requested amount being transferred is not always the amount received.

2.8.3 Unreachable code due to overly restrictive validation in `claimLidoWithdrawals()`

Technical Details

The `claimLidoWithdrawals()` function in `LidoARM.sol` contains defensive code to handle the edge case where Lido withdrawal request NFTs are transferred to the ARM contract from external addresses. However, this code is unreachable due to an earlier validation check.
At lines 151-160, the function validates each withdrawal request:

```
for (uint256 i = 0; i < requestIds.length; i++) {
    // Read the requested amount from storage
    uint256 requestAmount = lidoWithdrawalRequests[requestIds[i]];

    // Validate the request came from this Lido ARM contract and not
    // transferred in from another account.
    require(requestAmount > 0, "LidoARM: invalid request");

    totalAmountRequested += requestAmount;
}
```

The `require(requestAmount > 0)` check at line 157 ensures that only withdrawal requests stored in the `lidoWithdrawalRequests` mapping (i.e., those created by the ARM contract itself via `requestLidoWithdrawals()` or registered via `registerLidoWithdrawalRequests()`) can be claimed.
However, later in the function, there's defensive code meant to handle transferred withdrawal requests:

```
if (lidoWithdrawalQueueAmount < totalAmountRequested) {
    // This can happen if a Lido withdrawal request was transferred to the ARM contract
    lidoWithdrawalQueueAmount = 0;
} else {
    lidoWithdrawalQueueAmount -= totalAmountRequested;
}
```

The issue: If a withdrawal request NFT is transferred to the ARM contract from another address, it will not have an entry in the `lidoWithdrawalRequests` mapping. When attempting to claim it, `requestAmount` will be 0, causing the `require(requestAmount > 0)` check at line 157 to revert with "LidoARM: invalid request". Therefore, the defensive code at lines 163-166 can never be reached.

Impact

Low.

Recommendation

The code and comments should be aligned to reflect the actual behavior. There are two possible approaches:
Option 1: Remove the unreachable defensive code and update the comment to clarify that transferred requests cannot be claimed **Option 2**: If the intent is to support transferred withdrawal requests, remove the restrictive validation and implement proper handling

Developer Response

Option was was taken to remove the unreachable defensive code and update the comment to clarify that transferred requests cannot be claimed in PR #166.

2.9 Gas Savings Findings

2.9.1 Use custom errors instead of strings within `require` statements

Technical Details

The system uses literal strings within `require` statements which increases the deployed bytecode's size and represents a larger runtime gas cost for unhappy paths, given that the string literals must be loaded into memory.
Using custom errors within `require` statements mitigates both issues highlighted above, given that only the error's selector needs to be stored and loaded during reverts.

Impact

Gas optimization.

Recommendation

Use custom errors instead of string literals within `require` statements.

Developer Response

We started with string errors and have decided to keep them for consistency given the Lido ARM contract is live. There is also a cost in changing all the tests.
Reducing the contract size could be reason to push us to do this change in the future as the ARM contracts are pushing the limits of what can be deployed. Gas costs of unhappy paths is not a concern at the moment with sub 1 Gwei gas prices
Acknowledge but will not change

2.9.2 Storage read in `Abstract4626MarketWrapper.transferTokens` can be optimized

Technical Details

`Abstract4626MarketWrapper.transferTokens` may be called by the contract's owner to sweep tokens held by the contract. Given that it is assigned the `onlyOwner` modifier, the check at `Abstract4626MarketWrapper.sol#L241` may check `to` to either match `harvester` or `msg.sender`, which is already known to be the contract's owner.

Impact

Gas optimization.

Recommendation

Use `msg.sender` instead of `owner()` to avoid an unnecessary storage read.

Developer Response

Fixed in PR #164

2.10 Informational Findings

2.10.1 Unused immutable

Technical Details

`EtherFiARM.etherfiRedemptionManager` is initialized during the contract's deployment but never accessed in any of its methods.

Impact

Informational.

Recommendation

Remove the unused immutable variable to reduce the contract's deployed bytecode's size.

Developer Response

Fixed in PR #168

2.10.2 External protocols should not use ARM tokens as borrowable assets

Warning: This issue affects external protocols that might integrate ARM tokens, not the ARM protocol itself.

Technical Details

ARM contracts (LidoARM, EtherFiARM, OriginARM) are ERC4626 vaults that can be vulnerable to share price manipulation if used as borrowable assets in external lending protocols. An attacker can artificially inflate the ARM token price through direct asset donations, which instantly increases the value of their collateral without minting new shares, which can be used to perform an attack similar to the cream finance hack.
The vulnerability stems from how ARM calculates its total assets. The following code paths allow instant share price increases through direct donations:

1. **Market wrapper balance counting** - The `balanceOf()` function in Abstract4626MarketWrapper.sol returns lending market shares owned by the wrapper, which are included in the ARM's total assets.
2. **Reserve calculation** - The `getReserves()` function in AbstractARM.sol directly queries token balances.
3. **Total assets calculation** - The `_availableAssets()` function aggregates all asset sources.

Impact

Informational. The ARM protocol is not affected; only landing markets that integrate with ARM need to be aware of the limitations.

Recommendation

Consider adding prominent warnings in documentation and integration guides.

Developer Response

Acknowledged

2.11 Final Remarks

The Origin ARM protocol demonstrates a mature codebase with solid architectural design and appropriate security controls. The implementation follows established patterns and integrates well with external DeFi protocols. One high-severity issue was identified regarding the handling of losses from liquid staking token slashing events, which could result in unfair loss distribution among vault participants. This issue aside, the protocol exhibits strong security fundamentals with limited attack surface for malicious actors.