Pashov Audit Group

Ample Earn Security Review

December 12th 2025 - December 14th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

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. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Ample Earn

Ample Earn is a fork of Euler Earn, which is a fork of MetaMorpho that allocates deposits across up to 30 accepted ERC-4626 strategies, with changes such as zero-share protection, internal balance tracking to prevent share inflation, and removal of skim, ERC-2612 permit, and multicall functions. Users of EulerEarn are liquidity providers who want to earn from borrowing interest without having to actively manage the risk of their position. Deposits and withdrawals follow allocator-defined queues, with an option to keep funds in non-borrowable vaults for immediate availability.

5. Executive Summary

A time-boxed security review of the **layer3xyz/ample-earn** repository was done by Pashov Audit Group, during which **0xl33**, **newspace**, **0x37**, **Nyx** engaged to review **Ample Earn**. A total of **6** issues were uncovered.

Protocol Summary	
Project Name	Ample Earn
Protocol Type	Strategy Allocator
Timeline	December 12th 2025 - December 14th 2025

Review commit hash:
- ae9f3e117cac6024356249382ac91881247b52c9 (layer3xyz/ample-earn)

Fixes review commit hash:
- 18d3067d7f33c4bafef6a08333dc84253b06bee6 (layer3xyz/ample-earn)

Scope

`EulerEarn.sol` `EulerEarnFactory.sol` `AmpleEarn.sol` `AmpleEarnFactory.sol` `AmpleEarnReserve.sol` `AmpleEarnRouter.sol` `IAmpleEarn.sol` `IAmpleEarnFactory.sol` `IAmpleEarnReserve.sol` `IAmpleEarnRouter.sol` `AmpleErrorsLib.sol` `AmpleEventsLib.sol` `AmplePayoutLib.sol` `IEulerEarn.sol` `ConstantsLib.sol` `ErrorsLib.sol` `EventsLib.sol`

6. Findings

Findings count

Severity	Amount
Critical	1
Medium	1
Low	4
Total findings	**6**

Summary of findings

ID	Title	Severity	Status
[C-01]	Unrestricted router allows unauthorized merkle root setting	Critical	**Resolved**
[M-01]	`AmpleEarn.setMerkleRoots` can set incorrect `merkle` root	Medium	**Resolved**
[L-01]	Tolerant batch functions fail to handle calls to non-contract addresses	Low	Acknowledged
[L-02]	Missing manual vault registration function in factory	Low	Acknowledged
[L-03]	EVC operators can redirect payout funds to arbitrary addresses	Low	**Resolved**
[L-04]	Users may fail to withdraw because of the existing lost assets	Low	Acknowledged

Critical findings

[C-01] Unrestricted router allows unauthorized merkle root setting

Severity

Impact: High

Likelihood: High

Summary

AmpleEarnRouter.batchSetMerkleRootsStrict() is calling setMerkleRoots() directly without going through EVC. If the intention is to use the router as a payout manager, like in the tests (AmpleEarnRouterTest), this could allow anyone to set merkle roots for any vault and steal payouts.

Description

Inside AmpleEarn.setMerkleRoots(), caller authorization is based on _msgSenderOnlyEVCAccountOwner(), which returns msg.sender unless the call comes from the EVC:

```
function setMerkleRoots(
    uint256 totalTickets,
    uint8 designatedRecipientsCount,
    bytes32 designatedRecipientsRoot,
    bytes32 participantsRoot,
    VRFProofDetails calldata vrfProofDetails
) external nonReentrant returns (uint256 payoutId) {
    address msgSender = _msgSenderOnlyEVCAccountOwner();
    if (!isPayoutManager[msgSender] && msgSender != owner()) revert
AmpleErrorsLib.NotPayoutManagerRole();
    // ...
}
```

Because the router calls setMerkleRoots() directly (not via EVC), the vault's msgSender will be the router address, not the external caller. This causes problems like:

- If the router is not a payout manager on the vault, router calls revert with NotPayoutManagerRole().
- If the router is granted the payout manager role on a vault, then any external caller can call the router and successfully set merkle roots on that vault.

POC:

AmpleEarnForkTest.sol

```
function testFork_SetMerkleRootsWithDifferentAddress()
        public
        whenUserHasBalance(address(USDC), DEPOSITOR, 100e6)
        whenUserHasDeposit(DEPOSITOR, 100e6)
    {
        skip(7 days);
        vm.roll(block.number + 50400); // ~7 days of blocks
        _accrueInterest();
        address attacker = makeAddr("Attacker");

        // Generate merkle roots and total tickets.
        // NOTE: Use scoped blocks to avoid "stack too deep" in this large fork test.
        bytes32 merkleRootForParticipants;
        bytes32 merkleRootForDesignatedRecipients;
        uint256 totalTickets = 250e6; // 100 + 50 + 75 + 25
        {
            // Create user total tickets data for merkle tree
            TicketMerkleLeaf[] memory leaves = new TicketMerkleLeaf[](4);
            leaves[0] = TicketMerkleLeaf({user: attacker, totalTickets: 100e6, ticketStart: 0,
ticketEnd: 99e6});
            leaves[1] =
                TicketMerkleLeaf({user: SUPPLIER, totalTickets: 50e6, ticketStart: 100e6,
ticketEnd: 149e6});
            leaves[2] =
                TicketMerkleLeaf({user: RECEIVER, totalTickets: 75e6, ticketStart: 150e6,
ticketEnd: 224e6});
            leaves[3] =
                TicketMerkleLeaf({user: ONBEHALF, totalTickets: 25e6, ticketStart: 225e6,
ticketEnd: 249e6});

            DesignatedRecipientMerkleLeaf[] memory designatedRecipientLeaves = new
DesignatedRecipientMerkleLeaf[](4);
            designatedRecipientLeaves[0] =
                DesignatedRecipientMerkleLeaf({user: attacker, payoutAmount: 100e6,
designatedRecipientIndex: 0});
            designatedRecipientLeaves[1] =
                DesignatedRecipientMerkleLeaf({user: SUPPLIER, payoutAmount: 50e6,
designatedRecipientIndex: 1});
            designatedRecipientLeaves[2] =
                DesignatedRecipientMerkleLeaf({user: RECEIVER, payoutAmount: 75e6,
designatedRecipientIndex: 2});
            designatedRecipientLeaves[3] =
                DesignatedRecipientMerkleLeaf({user: ONBEHALF, payoutAmount: 25e6,
designatedRecipientIndex: 3});

            merkleRootForParticipants = MerkleHelper.generateMerkleRootForParticipants(leaves);
            merkleRootForDesignatedRecipients =

MerkleHelper.generateMerkleRootForDesignatedRecipients(designatedRecipientLeaves);
        }

        {

            vm.startPrank(attacker);
            SetMerkleRootsParams[] memory params = new SetMerkleRootsParams[](1);
            params[0] = SetMerkleRootsParams({
                vault: address(vault),
```

```
                participantsRoot: merkleRootForParticipants,
                designatedRecipientsRoot: merkleRootForDesignatedRecipients,
                designatedRecipientsCount: 4,
                totalTickets: totalTickets,
                vrfProofDetails: VRFProofDetails({
                    proof: bytes32(uint256(1)),
                    seed: bytes32(uint256(2)),
                    publicKey: bytes32(uint256(3)),
                    vrfHash: bytes32(uint256(4))
                })
            });

            uint256[] memory payoutIds = router.batchSetMerkleRootsStrict(params);

        }
    }
```

Recommendations

Enforce authorization inside batchSetMerkleRootsStrict() or call the setMerkleRoots() through EVC.

Medium findings

[M-01] `AmpleEarn.setMerkleRoots` can set incorrect `merkle` root

Severity

Impact: Medium

Likelihood: Medium

Description

`AmpleEarn.setMerkleRoots` sets the `designatedRecipientsRoot` and it contains the claim amount.

In `AmplePayoutLib.claimPayout`:

```
    uint256 payoutAmount = designatedRecipientLeaf.payoutAmount;
    bytes32 leaf = keccak256(
        abi.encode(designatedRecipientLeaf.user, payoutAmount,
designatedRecipientLeaf.designatedRecipientIndex)
    );
    if (!MerkleProof.verify(designatedRecipientProof, pool.designatedRecipientsRoot, leaf)) {
        revert AmpleErrorsLib.MerkleProofInvalid();
    }
```

When admin sets `setMerkleRoots` and any transaction that `deposit/redeem` runs before that transaction, it increases `balanceOf(PAYOUT_RESERVE)`, and `accruedInterestInPayoutReserve` is greater than expected (off-chain calculates claim amount using old `accruedInterestInPayoutReserve`)

Users can't claim excess interest and the interest is locked in contract.

Recommendations

Update `setMerkleRoots` to set `accruedInterestInPayoutReserve` directly.

Low findings

[L-01] Tolerant batch functions fail to handle calls to non-contract addresses

Both `AmpleEarnRouter.batchSetMerkleRootsTolerant()` and `AmpleEarnRouter.batchClaimPayoutTolerant()` fail to properly handle calls to non-contract addresses (EOAs or undeployed addresses), but with different failure modes:

batchSetMerkleRootsTolerant(): Uses high-level interface call syntax which automatically inserts an `extcodesize` check. If the target has no code, the call reverts before the try block executes, causing the catch block to never trigger and the entire batch operation to revert.

batchClaimPayoutTolerant(): Uses `evc.call()` which performs a low-level call. Low-level calls to EOAs succeed and return empty bytes. The try block succeeds, marking the operation as successful when no actual operation occurred. This results in false positives where users believe their claim succeeded when funds were never transferred.

Both behaviors break the intended "tolerant" design where individual failures should be caught, and the batch should continue processing remaining items.

Note: `batchClaimPayoutStrict()` also has the same issue as `batchClaimPayoutTolerant()`.

Proof of concept: Tests demonstrating these issues exist in `AmpleEarnRouterTest.test_batchSetMerkleRootsTolerant_invalidVault()` and `AmpleEarnRouterTest.test_batchClaimPayoutTolerant_invalidVault()`. To verify, uncomment the tests and run with the `-vvvv` flag to see the revert.

Note: before running `test_batchClaimPayoutTolerant_invalidVault()`, change from `payout` to `payoutAmount` in this line:

```
DesignatedRecipientMerkleLeaf memory designatedRecipientLeaf =
DesignatedRecipientMerkleLeaf({user: ALICE, payout: 100e6, designatedRecipientIndex: 0});
```

Recommendation: Add explicit validation that the vault address contains code before the try-catch blocks.

Example fix:

```
if (p.vault.code.length == 0) {
    results[i] = BatchResult({
        success: false,
        returnData: 0,
        errorData: abi.encodeWithSignature("InvalidVault(address)", p.vault)
    });
    emit SetMerkleRootsFailed(p.vault, msgSender,
```

```
abi.encodeWithSignature("InvalidVault(address)", p.vault));
    // or `ClaimPayoutFailed` for `batchClaimPayoutTolerant()`
    continue;
}
```

[L-02] Missing manual vault registration function in factory

The `AmpleEarnFactory.isStrategyAllowed()` function checks both `perspective.isVerified(id)` and `isVault[id]` to determine if a strategy is allowed. However, due to bytecode size limitations, AmpleEarn vaults cannot be deployed through `createAmpleEarn()` (based on audit documentation), which is the only function that populates the `isVault` mapping. Since vaults are deployed directly, `isVault` remains empty for all vaults.

This creates a single point of failure where strategy verification depends solely on `perspective.isVerified(id)`. The intended dual-check security mechanism (perspective OR factory registration) is non-functional, and the factory owner has no way to manually register trusted vaults.

Recommendation: Add a manual vault registration function.

[L-03] EVC operators can redirect payout funds to arbitrary addresses

Users who grant EVC operator permissions allow those operators to claim payouts on their behalf and redirect the funds to any address. When an operator calls `AmpleEarn.claimPayout()` through the EVC, the `_msgSender()` function returns the authenticated account (the user), but the operator controls the `to` parameter where funds are sent.

Attack flow:

- Alice grants Bob operator permissions via `EVC.setAccountOperator(Alice, Bob, true)`
- Bob calls `EVC.call(AmpleEarn, Alice, 0, claimPayout(..., to: Bob, ...))`
- In `AmpleEarn.claimPayout()`, `_msgSender()` returns Alice (authenticated account)
- Access check in `AmplePayoutLib.claimPayout()` passes: `Alice == designatedRecipientLeaf.user`
- Funds transferred to Bob's address instead of Alice.While granting operator permissions implies trust, users may authorize operators for specific purposes without realizing that it enables payout redirection.

Recommendation: Enforce that payouts go to the designated recipient specified in the merkle leaf.

Example fix:

```
function claimPayout(...) external nonReentrant {
    address msgSender = _msgSender();

    // Enforce payout goes to designated recipient
    if (to != designatedRecipientLeaf.user) {
        revert PayoutMustGoToDesignatedRecipient();
    }

    // Delegate to library
    AmplePayoutLib.claimPayout({...});
    ...
}
```

Note: This removes the flexibility for users to claim to different addresses, but enforces the merkle tree's intended recipient. Alternatively, if the flexibility is desired, clearly document that EVC operators can redirect payouts, and users should only grant operator permissions to fully trusted parties.

[L-04] Users may fail to withdraw because of the existing lost assets

AmpleEarn will distribute users' deposit into different strategy vaults to earn interest. When the strategy vault loses funds, the share price will not be impacted by the actual lose. This will cause the share price will be higher than the actual price. When users want to withdraw assets with the share price, users can fail to withdraw assets.

For example:

- Alice deposits 1000 assets, and Alice will get 1000 shares.
- The vault category loses 400 assets. So the real total assets are 600. The last total assets are 1000.
- Bob deposits 1000 assets, and Bob will get 1000 shares.
- Alice withdraws 1000 shares, she can get 1000 assets based on the current share price.
- Bob wants to withdraw all shares, the withdrawal will be reverted because there is not enough balance.

```
    function _accrueInterest() internal {
        // The total assets are not the actual real assets for this vault.
        (uint256 feeShares, uint256 newTotalAssets, uint256 newLostAssets) =
_accruedFeeAndAssets();
        _updateLastTotalAssets(newTotalAssets);
    }
    function _accruedFeeAndAssets()
        internal
        view
        returns (uint256 feeShares, uint256 newTotalAssets, uint256 newLostAssets)
    {
        // The assets that the Earn vault has on the strategy vaults.
```

```
        uint256 realTotalAssets;
        // All vaults must exist in this withdraw queue.
        for (uint256 i; i < withdrawQueue.length; ++i) {
            IERC4626 id = withdrawQueue[i];
            realTotalAssets += expectedSupplyAssets(id);
        }
        uint256 lastTotalAssetsCached = lastTotalAssets;
        // initially, the lostAssets is 0.
        // lastTotalAssetsCached - lostAssets is actually the last real total assets.
        if (realTotalAssets < lastTotalAssetsCached - lostAssets) {
            newLostAssets = lastTotalAssetsCached - realTotalAssets;
        } else {
            newLostAssets = lostAssets;
        }

@>        newTotalAssets = realTotalAssets + newLostAssets;
        uint256 totalInterest = newTotalAssets - lastTotalAssetsCached;
    }
```

Recommendations

Process the lost assets properly.