Formal Verification Report: Accountable Credit Vaults

- Repository: https://github.com/Cyfrin/audit-2025-09-accountable
- Audit Commit Hash: c163cec
- Fixes Commit Hash: 979c0eb
- Date: October 2025
- Author: @alexzoid_eth (@CyfrinAudits private formal verification engagement)
- Certora Prover version: 8.1.1

Table of Contents

About Accountable Credit Vaults

Accountable Credit Vaults establishes a permissioned credit infrastructure facilitating direct capital allocation between liquidity providers and verified borrowers. Built on asynchronous vault architecture conforming to ERC7540 specifications, the protocol enables withdrawal queue management and flexible credit strategies through modular loan administration components.

Formal Verification Approach

The verification environment tests both `FixedTerm` and `OpenTerm` strategy implementations. Global properties are validated across both configurations, while simpler properties are verified with `FixedTerm` only. Token interactions are simplified through CVL modeling. The verification suite systematically checks compliance with EIPs.

Assumptions

Assumptions are constraints applied during verification to make the problem tractable for the prover. They are classified as **Safe** (no impact on security guarantees) or **Unsafe** (may limit coverage).

Safe Assumptions

These assumptions reflect real-world constraints or simplify non-critical aspects without compromising verification validity:

- ERC20 tokens implemented in CVL, limited to 5 users per contract for tractability
- Block timestamp bounded to realistic values, block number non-zero
- Message sender assumed distinct from contracts under verification

Unsafe Assumptions

These assumptions reduce verification scope to avoid prover timeouts but potentially may miss edge cases:

- Loop unrolling capped at 3 iterations, bitwise operations overapproximated
- `reservedLiquidityBacked` proved only with `OpenTerm` due to discussion
- `securityAdmin()`, `operationsAdmin()` simplified into CVL ghosts
- `isVerified()` and `areVerified()` simplified into CVL ghosts
- `OpenTerm._processAvailableWithdrawals()` removed from verification due to prover issues

Formal Verification Methodology

Certora Formal Verification (FV) provides mathematical proofs of smart contract correctness by verifying code against a formal specification. It complements techniques like testing and fuzzing, which can only sometimes detect bugs based on predefined properties. In contrast, Certora FV examines all possible states and execution paths in a contract.

Simply put, the formal verification process involves crafting properties (similar to writing tests) in CVL language and submitting them alongside compiled Solidity smart contracts to a remote prover. This prover essentially transforms the contract bytecode and rules into a mathematical model and determines the validity of rules.

Types of Properties

When constructing properties in formal verification, we mainly deal with two types: **Invariants** and **Rules**.

Invariants

- Conditions that MUST **always remain true** throughout the contract's lifecycle.
- Process:
    1. Define an initial condition for the contract's state.
    2. Execute an external function.
    3. Confirm the invariant still holds after execution.
- Example: "Requests outside queue bounds must be empty"
- Use Case: Ensures **Valid State** properties - critical state constraints that MUST never be violated.
- Feature: Proven invariants can be reused in other properties with the `requireInvariant` keyword.

Rules

- Flexible checks for specific behaviors or conditions.
- Structure:
    1. Setup: Set assumptions (e.g., "Requests outside queue bounds must be empty").
    2. Execution: Simulate contract behavior by calling external functions.
    3. Verification:
        - Use `assert` to check if a condition is **always true** (e.g., "Queue should decrease by exactly the processed amount").
        - Use `satisfy` to verify a condition is **reachable** (e.g., "Queue pointer can advance past the empty head (prevents deadlock)").
- Example: "Any call to processUpToShares(amount) should decrease the total shares in the queue by that amount"
- Use Case: Verifies a broad range of properties, from simple state changes to complex business logic.

Verification Process

The process is divided into two stages: **Setup** and **Crafting Properties**.

Setup

This stage prepares the contract and prover for verification:

- Resolve external contract calls and dependencies.
- Simplify complex operations (e.g., math or bitwise calculations) for prover compatibility.
- Install storage hooks to monitor state changes.

- Address prover limitations (e.g., timeouts or incompatibilities).

Crafting Properties

This stage defines and implements the properties:

- Write properties in plain English for clarity.
- Categorize properties by purpose (e.g., Valid State, State Transition, EIP Compliance).
- Prove valid state invariants as a foundation for further rules

---

Verification Properties

The verification properties are categorized into two distinct types:

1. **Valid State (VS)**: System-wide invariants that MUST always hold true. These properties define the fundamental constraints of the protocol, such as accounting consistency and structural integrity. Once proven, these invariants serve as trusted assumptions in other properties via `requireInvariant`, reducing verification complexity.
2. **State Transition (ST)**: Properties that verify the correctness of transitions between valid states. Building upon the valid state invariants, these properties ensure the protocol's state machine operates correctly and that state changes are both authorized and sequentially valid.
3. **High Level (HL)**: Business logic and economic properties that cover the whole system from the users' point of view. Main focus on behavior of specific functions.
4. **EIP Compliance (EIP)**: Properties that verify the protocol's adherence to Ethereum Improvement Proposal standards.

Links to specific Certora Prover runs are provided for each property, with status indicators:

- ✅ Verified successfully
- ❌ Violated (indicates a potential issue)

Valid State

Valid State properties define the fundamental invariants that must always hold true throughout the protocol's lifecycle. These properties are organized by contract and proven as invariants, meaning they are checked to hold after every possible function execution from any valid initial state.

Property	Name	Description	Status	Notes
VS-01	`vaultCannotBeOwnerInOperator`	Vault cannot be an owner in operator relationships	✅	
VS-02	`operatorOwnerNonZero`	Zero address cannot be an owner in operator relationships	✅	
VS-03	`zeroAddressNoBalance`	Zero address cannot hold vault shares	✅	
VS-04	`vaultNoAllowances`	Vault never approves anyone to spend any tokens	✅	
VS-05	`vaultHoldsQueuedShares`	Vault's balance of its own shares must cover queued and redeemable shares	✅	
VS-06	`totalAssetsBackedByBalance`	Total assets tracked must be backed by vault's actual balance	✅	

Property	Name	Description	Status	Notes
VS-07	`reservedLiquidityBacked`	Reserved liquidity must not exceed total assets	❌	Issue: Reserved assets could be extracted from the Vault
VS-08	`zeroControllerEmptyState`	Zero address must have empty state for all vault fields	❌	Issue: Missing controller validation
VS-09	`depositStateConsistency`	If maxMint is zero, depositAssets must also be zero	✅	
VS-10	`mintPriceSetIfMintable`	If there are mintable shares, mint price must be set	✅	
VS-11	`pendingRedeemImpliesQueueRequest`	Pending redeem shares must have valid request in queue	✅	
VS-12	`pendingRedeemBacked`	Total pending redeem requests must equal totalQueuedShares	✅	
VS-13	`queueOrdering`	nextRequestId must be within queue bounds or one position after	✅	
VS-14	`queuePointersConsistency`	Queue pointers must be consistent (both zero or both non-zero)	✅	
VS-15	`validRequestIds`	All non-zero request IDs must be within valid queue bounds	✅	
VS-16	`requestIdControllerConsistency`	Controller's requestId must match request's controller field	✅	
VS-17	`withdrawalRequestControllerConsistency`	Request's controller must map back to same requestId	✅	
VS-18	`activeRequestsHaveShares`	Non-empty requests in queue must have non-zero shares	✅	
VS-19	`uniqueControllerRequestId`	A controller can have at most one active request ID	✅	
VS-20	`totalQueuedSharesMatchesRequests`	Total queued shares equals sum of all withdrawal requests	✅	
VS-21	`controllerRequestIdConsistency`	Queue boundary requests must maintain bidirectional mapping	✅	
VS-22	`sequentialRequestCounting`	Queue requests must be counted sequentially	✅	
VS-23	`instantRequestEmpty`	Instant request ID must have empty state	✅	
VS-24	`processedRequestsEmpty`	All processed requests must be completely empty	✅	
VS-25	`futureRequestsEmpty`	Requests outside queue bounds must be empty	✅	
VS-26	`pendingRedeemMatchesQueueShares`	Vault's pendingRedeemRequest must match queue shares	✅	
VS-27	`totalMaxWithdrawNotExceedReserved`	Total maxWithdraw cannot exceed reserved liquidity	❌	Issue: Instant fulfillRedeemRequest doesn't reserve liquidity

✅ All passed after fixes (local runs for `FixedTerm`: 1, 2, 3, 4, 5, 6; `OpenTerm`: 1, 2, 3, 4).

State Transitions

State Transition properties verify the correctness of transitions between valid states. These properties ensure that state changes occur only under the right conditions, such as calls to specific functions or time elapsing.

Property	Name	Description	Status	Notes
ST-01	`shareTransferMustBeToVerifiedAddress`	Share transfers must only go to verified addresses or the vault itself	❌	Issue: Bypass of transfer restrictions

✅ All passed after fixes.

High Level

These properties verify business logic and economic properties that cover the whole system from the users' point of view.

Property	Name	Description	Status	Notes
HL-01	`processUpToSharesDecreasesQueueShares`	Processing shares decreases queue by exact amount processed	✅	
HL-02	`processUpToRequestIdIncreasesNextId`	Processing requests advances nextRequestId appropriately	✅	
HL-03	`queueNotDeadlockOnEmptyEntry`	Queue can advance past empty entries to prevent deadlock	❌	Issue: Cancelling blocks withdrawal queue

✅ All passed after fixes (HL-01/02 and HL-03).

EIP20 Compliance

Prove contract is compatible with EIP20 (https://eips.ethereum.org/EIPS/eip-20).

Property	Name	Description	Status	Notes
EIP20-01	`eip20_totalSupplyIntegrity`	totalSupply() returns correct total token supply	✅	
EIP20-02	`eip20_balanceOfIntegrity`	balanceOf() returns correct balance for any account	✅	
EIP20-03	`eip20_allowanceIntegrity`	allowance() returns correct spending allowance	✅	
EIP20-04	`eip20_transferIntegrity`	transfer() correctly updates balances and maintains invariants	✅	
EIP20-05	`eip20_transferMustRevert`	transfer() reverts on insufficient balance or invalid addresses	✅	

Property	Name	Description	Status	Notes
EIP20-06	`eip20_transferSupportZeroAmount`	Zero amount transfers must be treated as normal transfers	❌	Issue: ERC20 zero amount transfer rejection
EIP20-07	`eip20_transferFromIntegrity`	transferFrom() correctly updates balances and allowances	✅	
EIP20-08	`eip20_transferFromMustRevert`	transferFrom() reverts on insufficient balance/allowance	✅	
EIP20-09	`eip20_transferFromSupportZeroAmount`	Zero amount transferFrom must be treated as normal	❌	Issue: ERC20 zero amount transfer rejection
EIP20-10	`eip20_approveIntegrity`	approve() correctly sets allowances without affecting balances	✅	
EIP20-11	`eip20_approveMustRevert`	approve() reverts on zero address operations	✅	

✅ All passed after fixes.

EIP4626 Compliance

Prove contract is compatible with EIP4626 (https://eips.ethereum.org/EIPS/eip-4626).

Property	Name	Description	Status	Notes
EIP4626-01	`eip4626_assetIntegrity`	Asset MUST be an EIP-20 token contract	✅	
EIP4626-02	`eip4626_assetMustNotRevert`	asset() MUST NOT revert	✅	
EIP4626-03	`eip4626_totalAssetsIntegrity`	Total assets MUST match tracked amount	✅	
EIP4626-04	`eip4626_totalAssetsMustNotRevert`	totalAssets() MUST NOT revert	✅	
EIP4626-05	`eip4626_convertToSharesMustNotDependOnCaller`	convertToShares MUST be caller-agnostic	✅	
EIP4626-06	`eip4626_convertToSharesMustNotRevert`	convertToShares MUST NOT revert on reasonable input	✅	
EIP4626-07	`eip4626_convertToSharesRoundDown`	convertToShares MUST round down	✅	
EIP4626-08	`eip4626_convertToAssetsMustNotDependOnCaller`	convertToAssets MUST be caller-agnostic	✅	
EIP4626-09	`eip4626_convertToAssetsMustNotRevert`	convertToAssets MUST NOT revert on reasonable input	✅	
EIP4626-10	`eip4626_convertToAssetsRoundDown`	convertToAssets MUST round down	✅	
EIP4626-11	`eip4626_maxDepositNoHigherThanActual`	maxDeposit MUST NOT exceed actual maximum	✅	

Property	Name	Description	Status	Notes
EIP4626-12	`eip4626_maxDepositDoesNotDependOnUserBalance`	maxDeposit MUST NOT rely on user balance	✅	
EIP4626-13	`eip4626_previewDepositNoMoreThanActualShares`	previewDeposit MUST NOT exceed actual shares	✅	
EIP4626-14	`eip4626_previewDepositMustIgnoreLimits`	previewDeposit MUST ignore deposit limits	✅	
EIP4626-15	`eip4626_previewDepositMustIncludeFees`	previewDeposit MUST include fees	✅	
EIP4626-16	`eip4626_previewDepositMustNotDependOnCaller`	previewDeposit MUST be caller-agnostic	✅	
EIP4626-17	`eip4626_previewDepositMayRevertOnlyWithDepositRevert`	previewDeposit MAY revert only if deposit would	✅	
EIP4626-18	`eip4626_depositIntegrity`	deposit() MUST mint exact shares for assets	✅	
EIP4626-19	`eip4626_depositRespectsApproveTransfer`	deposit() MUST respect ERC20 allowances	✅	
EIP4626-20	`eip4626_depositMustRevertIfCannotDeposit`	deposit() MUST revert if cannot transfer all assets	✅	
EIP4626-21	`eip4626_maxMintNoHigherThanActual`	maxMint MUST NOT exceed actual maximum	✅	
EIP4626-22	`eip4626_maxMintDoesNotDependOnUserBalance`	maxMint MUST NOT rely on user balance	✅	
EIP4626-23	`eip4626_maxMintZeroIfDisabled`	maxMint MUST return 0 if mints disabled	✅	
EIP4626-24	`eip4626_previewMintNoFewerThanActualAssets`	previewMint MUST NOT underestimate assets needed	✅	
EIP4626-25	`eip4626_previewMintMustIgnoreLimits`	previewMint MUST ignore mint limits	✅	
EIP4626-26	`eip4626_previewMintMustIncludeFees`	previewMint MUST include fees	✅	
EIP4626-27	`eip4626_previewMintMustNotDependOnCaller`	previewMint MUST be caller-agnostic	✅	
EIP4626-28	`eip4626_previewMintMayRevertOnlyWithMintRevert`	previewMint MAY revert only if mint would	✅	
EIP4626-29	`eip4626_mintIntegrity`	mint() MUST mint exact shares for assets	✅	
EIP4626-30	`eip4626_mintRespectsApproveTransfer`	mint() MUST respect ERC20 allowances	✅	
EIP4626-31	`eip4626_mintMustRevertIfCannotMint`	mint() MUST revert if cannot mint exact shares	✅	
EIP4626-32	`eip4626_maxWithdrawNoHigherThanActual`	maxWithdraw MUST NOT exceed actual maximum	❌	Issue: Invalid maxWithdraw() check in withdraw()
EIP4626-33	`eip4626_maxWithdrawZeroIfDisabled`	maxWithdraw MUST return 0 if withdrawals disabled	✅	
EIP4626-34	`eip4626_maxWithdrawMustNotRevert`	maxWithdraw MUST NOT revert	✅	
EIP4626-35	`eip4626_withdrawIntegrity`	withdraw() burns shares and sends exact assets	✅	
EIP4626-36	`eip4626_withdrawMustRevertIfCannotWithdraw`	withdraw() MUST revert if cannot transfer assets	✅	

Property	Name	Description	Status	Notes
EIP4626-37	`eip4626_maxRedeemNoHigherThanActual`	maxRedeem MUST NOT exceed actual maximum	✅	
EIP4626-38	`eip4626_maxRedeemZeroIfDisabled`	maxRedeem MUST return 0 if redemption disabled	✅	
EIP4626-39	`eip4626_maxRedeemMustNotRevert`	maxRedeem MUST NOT revert	✅	
EIP4626-40	`eip4626_redeemIntegrity`	redeem() burns exact shares and sends assets	✅	

✅ All passed after fixes.

EIP7540 Compliance

Prove contract is compatible with EIP7540 (https://eips.ethereum.org/EIPS/eip-7540) - Asynchronous ERC-4626 Tokenized Vaults.

Property	Name	Description	Status	Notes
EIP7540-01	`eip7540_previewWithdrawMustRevert`	previewWithdraw MUST revert for all callers and inputs	✅	
EIP7540-02	`eip7540_previewRedeemMustRevert`	previewRedeem MUST revert for all callers and inputs	✅	
EIP7540-03	`eip7540_requestRedeemMustRemoveSharesFromOwner`	Shares MUST be removed from owner custody on requestRedeem	✅	
EIP7540-04	`eip7540_requestRedeemMustRevertIfCannotRequest`	requestRedeem MUST revert if shares cannot be requested	✅	
EIP7540-05	`eip7540_requestRedeemMustRespectOwnerOrOperator`	Owner MUST be msg.sender or have approved operator	✅	
EIP7540-06	`eip7540_redeemControllerMustBeCallerOrOperator`	Controller MUST be msg.sender or have approved operator	✅	
EIP7540-07	`eip7540_pendingRedeemRequestMustNotRevert`	pendingRedeemRequest MUST NOT revert on valid input	✅	
EIP7540-08	`eip7540_pendingRedeemRequestIndependentOfCaller`	pendingRedeemRequest MUST NOT vary by caller	✅	
EIP7540-09	`eip7540_claimableRedeemRequestIndependentOfCaller`	claimableRedeemRequest MUST NOT vary by caller	✅	
EIP7540-10	`eip7540_claimableRedeemRequestMustNotRevert`	claimableRedeemRequest MUST NOT revert on valid input	✅	
EIP7540-11	`eip7540_setOperatorMustReturnTrue`	setOperator MUST return True	✅	
EIP7540-12	`eip7540_setOperatorMustSetStatus`	setOperator MUST set the operator status to approved value	✅	
EIP7540-13	`eip7540_withdrawControllerMustBeCallerOrOperator`	Withdraw controller MUST be msg.sender or have approved operator	✅	
EIP7540-14	`eip7540_requestIdZeroUsesControllerOnly`	When requestId==0, MUST use controller to discriminate state	✅	
EIP7540-15	`eip7540_requestIdConsistentlyZero`	If any requestId is 0, all MUST be 0	✅	
EIP7540-16	`eip7540_requestMustNotSkipClaimableState`	Request MUST NOT skip the Claimable state	✅	

✅ All passed before and after fixes.

Real Issues Properties

This section documents vulnerabilities discovered during the manual audit (including issues by all participants) and formal verification process. Each issue demonstrates how formal properties detected the vulnerability and confirmed its resolution after applying the fix.

[CRITICAL] Cancelling redeem requests permanently blocks the withdrawal queue (#9)

`AccountableWithdrawalQueue` can deadlock at the head if the current head entry (`_queue.nextRequestId`) is fully removed (e.g., by a cancel that zeroes `shares` and clears `controller`) without advancing `nextRequestId`.

❌ Violated: https://prover.certora.com/output/52567/57215c2865ee4ea99cef925f2426da37/?anonymousKey=abf3eabb5d6b0dbcf85266f08069455487c7653a

```
// processUpToShares can advance past an empty head entry, preventing queue deadlock
rule queueNotDeadlockOnEmptyEntry(env e) {

    setupValidState(e);

    mathint nextIdBefore = ghostQueueNextRequestId128;
    mathint lastIdBefore = ghostQueueLastRequestId128;

    require(nextIdBefore > 0 && nextIdBefore < lastIdBefore,
       "Require at least 2 requests in queue (head + one more)");

    // Require that head request is empty (controller == address(0)), as happens after a
cancel
    require(ghostQueueRequestsController[nextIdBefore] == 0,
       "Require that head request is empty");

    processUpToShares(e, max_uint256);

    // Verify that the queue pointer can advance past the empty head (prevents deadlock)
    satisfy(ghostQueueNextRequestId128 > nextIdBefore);
}
```

✅ Passed with final fixes: https://prover.certora.com/output/52567/c4a4056756af43a794d7c87bf36297a0/?anonymousKey=2c1a4745a7e5d42b8b84394d01ac7b78e8e43b49

[MEDIUM] Complete bypass of transfer restrictions on vault share token is possible (#4)

In `AccountableVault.sol` (which is inherited by the `AccountableAsyncRedeemVault`, we have certain transfer restrictions (KYC, if from address is subject to a throttle timestamp), applied in `_checkTransfer()` function.

❌ Violated (in `claimCancelRedeemRequest`): https://prover.certora.com/output/52567/b953632aecec42f8bddaf4ebb2a74471/?anonymousKey=37843116e90548773ab763348c7c7f059d4760fa

```
// Any share transfer must go to a verified address or the vault itself
// This catches cases where internal _transfer bypasses the _checkTransfer validation
rule shareTransferMustBeToVerifiedAddress(env e, method f, address to)
    filtered { f -> !EXCLUDED_FUNCTION(f) }
{
    setupValidState(e);

    mathint balanceBefore = ghostERC20Balances128[_Vault][to];
    mathint totalSupplyBefore = ghostERC20TotalSupply256[_Vault];

    calldataarg args;
    f(e, args);

    mathint balanceAfter = ghostERC20Balances128[_Vault][to];
    mathint totalSupplyAfter = ghostERC20TotalSupply256[_Vault];

    // Check if address is verified after the transaction
    bool isVerified = ghostAllowed[to];

    // If this is a mint operation, total supply would increase
    bool isMintOperation = totalSupplyAfter > totalSupplyBefore;

    // If balance increased via transfer (not minting), recipient must be verified or the
vault
    assert(balanceAfter > balanceBefore && !isMintOperation =>
           (isVerified || to == _Vault),
           "Share transfers must only go to verified addresses or the vault");
}
```

✅ Passed with final fixes: https://prover.certora.com/output/52567/68959381f803443db5ccbd1fad0d1d56/?anonymousKey=1ea64a34ed56bea4c0eb5f7bc175de9a73537277

[MEDIUM] Manual/Instant `fulfillRedeemRequest` doesn't reserve liquidity (#13)

Manual fulfillment paths (`fulfillRedeemRequest`) and the instant branch of `requestRedeem` mark shares as claimable without increasing `reservedLiquidity`.

❌ Violated: https://prover.certora.com/output/52567/af600209eef7404080e25e0ecc70589f/?anonymousKey=7b51592395a9f6ec802c89f89c0d789bb76540c9

```
// Total maxWithdraw across all users cannot exceed reserved liquidity
invariant totalMaxWithdrawNotExceedReserved(env e)
    TOTAL_MAX_WITHDRAW_ASSETS() <= ghostReservedLiquidity256
```

✅ Verified after fixes: https://prover.certora.com/output/52567/00ba79bba8064665a1a2df62bfc3e74e/?anonymousKey=50770252acecca136ae5ace305573e483bde2eb1

[MEDIUM] Invalid `maxWithdraw()` check in `withdraw()` (#33)

Vault incorrectly checks `maxWithdraw(receiver)` instead of `maxWithdraw(controller/owner)`.

❌ A property is violated: https://prover.certora.com/output/52567/ef88bd2d76b74cafb175f8d026e484b3/?anonymousKey=599db11fbc5df1632ff4006c69a03f836b23fa6c

```
// MUST NOT be higher than the actual maximum that would be accepted
rule eip4626_maxWithdrawNoHigherThanActual(env e, uint256 assets, address receiver, address
owner) {

    setup(e);

    storage init = lastStorage;

    mathint limit = maxWithdraw(e, owner) at init;

    withdraw@withrevert(e, assets, receiver, owner) at init;
    bool reverted = lastReverted;

    // Withdrawals above the limit must revert
    assert(assets > limit => reverted, "Withdraw above limit MUST revert");
}
```

✅ Passed after fixes: https://prover.certora.com/output/52567/8e7cfdf612d64a4cb7e5d9d9d939968e/?anonymousKey=a961467ded443bd1cab3718ca882be71f38887e9

[LOW] Reserved assets could be extracted from the Vault (#27)

Some strategy functions can release assets without checking if those assets are part of `reservedLiquidity`.

❌ Violated (in `OpenTerm`): https://prover.certora.com/output/52567/4fbec9433ca24d3999cbb10f3a16d213/?anonymousKey=cb5dca2ddcd5ad781c719f6aae4051f9846085ad

```
// Reserved liquidity must not exceed total assets
invariant reservedLiquidityBacked(env e)
    ghostReservedLiquidity256 <= ghostTotalAssets256
```

✅ Verified after fixes (in `OpenTerm`): https://prover.certora.com/output/52567/6c23fc5e692e4f6b81a27bb662599293/?anonymousKey=154591958effceb15daabd64f146f81fcb361bd6

Excluded from the `FixedTerm` configuration due to discussion.

[LOW] Missing controller validation in `AccountableAsyncRedeemVault::requestRedeem` allows zero address state (#24)

The `requestRedeem()` function fails to call `_checkController(controller)` validation, allowing the zero address to accumulate vault state.

❌ A property is violated: https://prover.certora.com/output/52567/acc42433123e4b289c0f84e69fa52a44/?anonymousKey=e60b3d66b5574868073bfde4218b385aa2fe5f2a

```
// VS-08: Zero address must have empty state for all vault fields
invariant zeroControllerEmptyState(env e)
    ghostVaultStatesMaxMint256[0] == 0 &&
    ghostVaultStatesMaxWithdraw256[0] == 0 &&
    ghostVaultStatesDepositAssets256[0] == 0 &&
    ghostVaultStatesRedeemShares256[0] == 0 &&
    ghostVaultStatesDepositPrice256[0] == 0 &&
    ghostVaultStatesMintPrice256[0] == 0 &&
    ghostVaultStatesRedeemPrice256[0] == 0 &&
    ghostVaultStatesWithdrawPrice256[0] == 0 &&
    ghostVaultStatesPendingRedeemRequest256[0] == 0 &&
    ghostRequestIds128[0] == 0
filtered { f -> !EXCLUDED_FUNCTION(f) } { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

✅ Passed after fixes: reports/fixed_valid_state_48.html

[INFO] ERC20 zero amount transfer rejection (#25)

The `_checkTransfer` function reverts on zero-amount transfers, violating ERC-20 standard which mandates that transfers of 0 values MUST be treated as normal transfers.

❌ A property is violated: https://prover.certora.com/output/52567/9c9c3c73f4d64f9baf1284ced4f4a8f5/?anonymousKey=160f0b0d10e3f688f1981708e4aa3819e7023a80

```
// EIP20-06: Verify transfer() handles zero amount transfers correctly
// EIP-20: "Transfers of 0 values MUST be treated as normal transfers and fire the Transfer
event."
rule eip20_transferSupportZeroAmount(env e, address to, uint256 amount) {

    setup(e);

    // Perform transfer
    transfer(e, to, amount);

    // Zero amount transfers must succeed
    satisfy(amount == 0);
}

// EIP20-09: Verify transferFrom() handles zero amount transfers correctly
// EIP-20: "Transfers of 0 values MUST be treated as normal transfers and fire the Transfer
event."
```

```
rule eip20_transferFromSupportZeroAmount(env e, address from, address to, uint256 amount) {

    setup(e);

    // Perform the transferFrom
    transferFrom(e, from, to, amount);

    // Zero amount transferFrom must succeed
    satisfy(amount == 0);
}
```

✅ Passed after fixes: https://prover.certora.com/output/52567/e0230dd1a8b44d4e8df7111b5e66741f/?anonymousKey=f87c2890f3a5cf744c08c20e7b05461c3eba735d

Setup and Execution Instructions

For step-by-step installation steps refer to this setup tutorial.

Required source code modifications

No modifications are required to the source code for Certora verification. The verification setup uses harness contracts that wrap the original contracts to enable formal verification without modifying production code.

Verification Execution

Running Verifications

1. **Valid State Properties (Fixed-Term Strategy)**:

```
# Run all valid state invariants for fixed-term strategy
certoraRun certora/confs/fixed/fixed_valid_state.conf
```

2. **Valid State Properties (Open-Term Strategy)**:

```
# Run all valid state invariants for open-term strategy
certoraRun certora/confs/open/open_valid_state.conf
```

3. **State Transition Properties**:

```
# Run state transition verification
certoraRun certora/confs/state_transition.conf
```

4. **High-Level Business Logic**:

```
# Run high-level properties verification
certoraRun certora/confs/high_level.conf
```

5. **EIP Compliance Verification**:

```
# Run EIP-20 compliance checks
certoraRun certora/confs/eip20_compliance.conf

# Run EIP-4626 tokenized vault compliance
certoraRun certora/confs/eip4626_compliance.conf

# Run EIP-7540 async vault compliance
certoraRun certora/confs/eip7540_compliance.conf
```

Advanced Options

To optimize verification time or debug issues, you can run specific rules:

1. **Run with specific rule**:

```
certoraRun certora/confs/fixed/fixed_valid_state.conf --rule vaultHoldsQueuedShares
certoraRun certora/confs/high_level.conf --rule queueNotDeadlockOnEmptyEntry
```

2. **Run with specific external function**:

```
certoraRun certora/confs/state_transition.conf --method
"requestRedeem(uint256,address,address)"
certoraRun certora/confs/eip4626_compliance.conf --method "deposit(uint256,address)"
```