yAudit

August 2026

Prepared for
Twyne

Audited by
ret2basic
HHK
adriro

Twyne - Morpho Vault

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Twyne is a credit delegation protocol that lets borrowers rent unused borrowing power from other lenders to boost their Liquidation LTV. Lenders earn additional yield while borrowers get to ramp up their leverage or insulate their debt.

1.2 Audit Scope

This audit covers protocol updates since version 1.0.6, including the new integration with Morpho, across 3 days of review.

```
src
├── Periphery
│   └── AssetZap.sol
├── TwyneFactory
│   └── CollateralVaultFactory.sol
├── operators
│   ├── MorphoDeleverageOperator.sol
│   ├── MorphoLeverageOperator.sol
│   ├── MorphoTeleportOperator.sol
│   ├── AaveV3LeverageOperator.sol
│   ├── LeverageOperator.sol.sol
│   └── handlers
│       └── WstethHandler.sol
└── twyne
    ├── AaveV3CollateralVault.sol
    ├── CollateralVaultBase.sol
    ├── EulerCollateralVault.sol
    ├── MorphoCollateralVault.sol
    └── VaultManager.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	0
Medium	1
Low	3
Informational	2

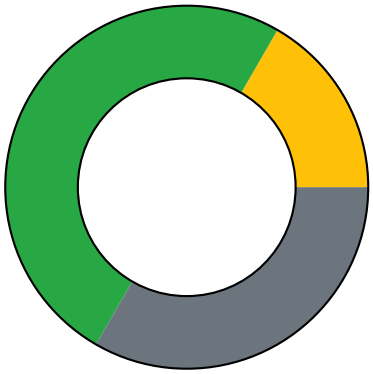

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

The review of Twyne's Morpho integration and broader protocol updates identified no critical or high-severity vulnerabilities, supporting the overall soundness of the codebase. One medium-severity issue in the Morpho leverage flow allowed user collateral to mask insufficient

swap output, while the low-severity findings centered on emergency pause behavior, integration configuration assumptions, and accounting for unused swap inputs. The team promptly addressed the issues raised during the engagement, demonstrating strong responsiveness throughout the review.

2 Audit Overview

2.1 Project Information

Protocol Name: Twyne
Repository: https://github.com/0xTwyne/twyne-contracts
Commit Hashes:

- Initial revision: d3cee306a9018ea0f405c8da9db79ed34487ae9e
- Borrower EVC only owner fix: 55cfe26aae63be54f9e58196bcbac7b1a2672b59
- PR #297: 21c25f45fd1a04f0fac6904f09decbed5e5decb7

Final Commit Hashes:

- Private repository: 307c5b9ad74b6cc922e78a473e7ec8e28f5e9a84
- Public repository: beb4bf15a70b1a8abcf7e17e8a08f556ee5f84ab

2.2 Audit Timeline

The audit was conducted from August 17 to 19, 2026.

2.3 Audit Resources

- Code repositories and documentation

2.4 Critical Findings

None.

2.5 High Findings

None.

2.6 Medium Findings

2.6.1 User collateral masks insufficient swap output in Morpho leverage

`MorphoLeverageOperator` includes the user's collateral contribution when validating the minimum swap output. A swap can therefore return less collateral than the user requires while the operation succeeds and leaves the user with the full flash-loan-sized debt.

Technical Details

`executeLeverage()` transfers `userCollateralAmount` from the borrower to the operator before requesting the Morpho flash loan. After the loan token is swapped, `onMorphoFlashLoan()` validates `minAmountOut` against the operator's entire collateral-token balance:

```
uint collateralBalance = IERC20(collateralToken).balanceOf(address(this));
require(collateralBalance >= minAmountOut, T_SlippageCheckFailed());
```

The checked balance includes the user's contribution and the current swap output. It does not isolate the amount produced by the swap, even though `minAmountOut` is documented as the minimum collateral token amount expected from the swap. For example, if the user contributes 10 collateral tokens, requires 10 tokens from the swap, and the swap returns only 1 token, the check passes against the aggregate balance of 11 tokens.

The operator then transfers the aggregate collateral balance to the vault and borrows the flash-loan amount to repay Morpho. If the user's contribution is sufficient to keep the final position within its borrow limit, an adversely executed swap can bypass the user's minimum and leave the user with debt backed primarily by their original collateral. This issue exists even when the swap consumes the entire flash-loan input because the defect is how collateral output is validated.

Impact

Medium. A stale, malformed, or manipulated swap can execute below the user's operator-level minimum without reverting. The borrower can incur debt equal to the flash-loan amount while receiving little or no collateral from the swap, with the loss bounded by the debt their contributed collateral can support.

Recommendation

Snapshot the collateral balance immediately before executing the swap and validate only the balance increase:

```
uint collateralBefore = IERC20(collateralToken).balanceOf(address(this));
ISwapper(SWAPPER).multicall(swapData);
uint swapOutput = IERC20(collateralToken).balanceOf(address(this)) - collateralBefore;
require(swapOutput >= minAmountOut, T_SlippageCheckFailed());
```

Alternatively, collect the user's collateral only after validating swap output, following the Aave operator's ordering.

Developer Response

Fixed in PR #294

2.7 Low Findings

2.7.1 Zero-value withdrawals reserve fresh CLP liquidity while Frozen

Technical Details

The `PauseState.Frozen` state permits borrowers to call `repay()`, `withdraw()`, and `redeemUnderlying()` to wind down their positions. Each of these functions passes the `whenNotPaused(PauseState.Paused)` guard and unconditionally invokes `_handleExcessCredit(_invariantCollateralAmount())` after completing its nominal operation. The `_handleExcessCredit` implementations in `AaveV3CollateralVault.sol:135-150`, `EulerCollateralVault.sol:84-105`, and `MorphoCollateralVault.sol:101-125` are bidirectional: when tracked collateral is below the computed invariant target, the function calls `intermediateVault.borrow()` to reserve additional credit and places that credit into the external integration.

The invariant calculation in `_collateralScaledByLiqLTV1e8` includes `intermediateVault.cash()` when `isRebalancing` is true. A position can be under-reserved when the intermediate vault previously had limited cash, constraining the dynamic reservation leg. If new CLP deposits subsequently make cash available, the borrower's next unwind call recomputes a higher invariant target. Since no parameter or state check prevents the reserve branch from executing while Frozen, the bidirectional hook borrows the difference from the intermediate vault. In Euler, this leaves the borrowed eWETH in the collateral vault's balance; in Aave, `rebalanceATokens_CV` transfers the assets into aTokens; in Morpho, `__supplyCollateral` deploys them as market collateral.

A borrower can trigger this behavior with a zero-value `withdraw(0)` call, which completes without requiring additional capital or external-debt repayment. The pause check only gates the entry function; it does not constrain the internal reserve operation. As a result, the freeze does not reliably cap protocol exposure at the moment of the emergency action.

Impact

Low. The verified call consumes previously withdrawable intermediate-vault cash, creates new debt for the collateral vault, and deploys the CLP assets as external collateral. This impairs CLP withdrawal capacity and increases the principal exposed to the condition that prompted the freeze. While the reserved assets remain protocol-accounted collateral and an immediate token loss was not demonstrated, the behavior defeats the intended emergency exposure containment. If the incident develops into an external liquidation or integration loss, the newly reserved CLP collateral can be seized to satisfy the borrower's pre-existing external debt, transferring fresh CLP capital into the frozen integration after governance has acted to prevent new exposure.

Recommendation

Make excess-credit handling release-only whenever the factory is Frozen. Permit repayment of existing intermediate-vault debt but skip the `intermediateVault.borrow` branch. Keep bidirectional reservation only in Active, and let unwind calls revert on final health checks if they cannot proceed without fresh credit. This retains wind-down functionality without allowing fresh CLP exposure.

```
1   function _handleExcessCredit(uint __invariantCollateralAmount) internal
  override {
```

```
        uint vaultAssets = totalAssetsDepositedOrReserved;
        if (vaultAssets > __invariantCollateralAmount) {
            uint excess = Math.min(vaultAssets - __invariantCollateralAmount,
    intermediateVault.debtOf(address(this)));
            totalAssetsDepositedOrReserved = vaultAssets - intermediateVault.
    repay(excess, address(this));
    -     } else if (vaultAssets < __invariantCollateralAmount) {
    +     } else if (vaultAssets < __invariantCollateralAmount &&
    collateralVaultFactory.pauseState() == PauseState.Active) {
            totalAssetsDepositedOrReserved = vaultAssets + intermediateVault.
    borrow(__invariantCollateralAmount - vaultAssets, address(this));
        }
    }
```

Apply the same guard to the corresponding `_handleExcessCredit` implementations in Aave and Morpho collateral vaults.

Developer Response

Acknowledged.

2.7.2 Morpho leverage over-borrows when swaps return unused loan tokens

`MorphoLeverageOperator` always borrows the full flash-loan amount after a leverage swap, even when the swap returns unused loan tokens to the operator. The borrower can incur unnecessary debt while tokens from their operation remain stranded in the shared operator.

Technical Details

`onMorphoFlashLoan()` transfers the full flash-loan amount to the Swapper and executes the caller-provided route. An exact-output, partially filled, or refunding route can return unconsumed loan tokens to `MorphoLeverageOperator` together with collateral output that satisfies `minAmountOut`. The defect is therefore independent of collateral-output validation and instead concerns how much the vault borrows after the swap.

The callback does not measure returned loan tokens. It always requests the original flash-loan amount from the collateral vault:

```
data: abi.encodeCall(CollateralVaultBase.borrow, (amount, address(this)))
```

If a 100-token flash loan uses only 70 tokens and returns 30, the operator still borrows 100 tokens from the vault. Morpho then pulls 100 tokens to settle the flash loan, leaving the returned 30 tokens in the operator while the borrower owes the full 100-token debt. These tokens were produced by the current operation rather than being a preexisting donation.

Impact

Low. A refunding or partial-fill route can leave the borrower with debt exceeding the amount required to complete the leverage operation. The unused portion remains in the operator and can subsequently be treated as a donation, causing a loss to the borrower who originated it.

Recommendation

Snapshot the operator's loan-token balance immediately before executing the swap and measure the current call's returned amount. Apply returned loan tokens toward flash-loan repayment and borrow only the remaining shortfall. After Morpho repayment, sweep the operator's entire remaining loan-token balance to the authenticated borrower, mirroring the deleverage operator.

For example:

```
 function executeLeverage(...) external nonReentrant {
     ...
     MORPHO.flashLoan(...);

+    uint loanTokenBalance = IERC20(mp.loanToken).balanceOf(address(this));
+    if (loanTokenBalance > 0) {
+        IERC20(mp.loanToken).safeTransfer(msgSender, loanTokenBalance);
+    }
 }

 function onMorphoFlashLoan(uint amount, bytes calldata data) external {
     ...
     IERC20(loanToken).safeTransfer(SWAPPER, amount);
+    uint loanBalanceBeforeSwap = IERC20(loanToken).balanceOf(address(this));
     ISwapper(SWAPPER).multicall(swapData);
+    uint returnedLoan = IERC20(loanToken).balanceOf(address(this)) -
loanBalanceBeforeSwap;
+    uint borrowAmount = returnedLoan >= amount ? 0 : amount - returnedLoan;
     ...
-    data: abi.encodeCall(CollateralVaultBase.borrow, (amount, address(this)))
+    data: abi.encodeCall(CollateralVaultBase.borrow, (borrowAmount, address(
this)))
 }
```

Developer Response

Fixed in PR #294

2.7.3 Upgrading a paused factory silently restores the Active state

Upgrading a paused `CollateralVaultFactory` resets its effective pause state to `PauseState.Active`. Operations that governance intended to keep disabled can therefore resume after the upgrade.

Technical Details

The previous factory implementation inherits `PausableUpgradeable`, which stores its `_paused` boolean in OpenZeppelin's ERC-7201 namespaced storage. The new implementation removes this inheritance and instead declares `PauseState public pauseState` in the factory's linear storage.

No reinitializer or migration translates the legacy `_paused` value into the new enum. Consequently, `pauseState` reads its default value of zero, `PauseState.Active`, even when the proxy was paused before the upgrade. The documented upgrade payload invokes `setAdmin()` through `upgradeToAndCall()`, but does not preserve or reapply the pause state. After the collateral vault beacons are upgraded to implementations that query `pauseState()`, functions protected by `whenNotPaused()` become available without an explicit unpause action.

Impact

Low. The issue requires governance to upgrade the factory while it is paused, and the admin can restore the intended state with another transaction. However, the unexpected activation creates a window in which operations disabled as part of an emergency response can resume.

Recommendation

Add a one-time upgrade migration that reads the legacy namespaced `_paused` value and initializes `pauseState` accordingly. Preserve the legacy storage definition until migration is complete, and make the migration callable within the owner-authorized `upgradeToAndCall` payload. At minimum, the deployment transaction must atomically call `setPauseState(PauseState.Paused)` whenever the pre-upgrade factory was paused, or ensure the pause is not enabled.

Developer Response

Acknowledged. The upgrade will be done in a normal state.

2.8 Gas Savings Findings

2.8.1 Cache `maxTwyneLTV` during external liquidation splitting

The external liquidation paths redundantly fetch the effective max Twyne LTV while splitting the remaining collateral.

Technical Details

`AaveV3CollateralVault.splitCollateralAfterExtLiq()`, `EulerCollateralVault.splitCollateralAfterExtLiq()`, and `MorphoCollateralVault.splitCollateralAfterExtLiq()` each call `twyneVaultManager.maxTwyneLTVs(...)` to calculate `userCollateral`. That manager lookup reads the configured max-LTV and ramp metadata before evaluating the current ramped value.

Each corresponding `handleExternalLiquidation()` calls `createVaultSnapshot()` before invoking `splitCollateralAfterExtLiq()`. The snapshot already stores the effective liquidation parameters, including max Twyne LTV, in transient `_maxTwyneLiqLTV`, and `_liqParams()` returns the cached values while the snapshot is active.

Impact

Gas Savings.

Recommendation

Use the already-cached `_maxTwyneLiqLTV` in all three `splitCollateralAfterExtLiq()` implementations, or pass that cached value into the helper explicitly.

Developer Response

Fixed in PR #299.

2.8.2 Duplicate intermediate vault validation

Each collateral vault creation function validates the intermediate vault before `_deployVault()` repeats the same check. Successful vault deployments therefore pay for an unnecessary external call to `VaultManager`.

Technical Details

`createEulerCollateralVault()`, `createAaveV3CollateralVault()`, and `createMorphoCollateralVault()` each call `vaultManager.isIntermediateVault(_intermediateVault)` and revert when it returns false. After their protocol-specific validation, all three functions invoke `_deployVault()`.

`_deployVault()` performs the identical `isIntermediateVault()` call again before deploying the beacon proxy. Since every caller reaches the shared helper and the relevant state cannot change between these synchronous checks, the second validation does not provide additional protection. It only adds the gas cost of another external view call and associated return-data handling to every successful collateral vault deployment.

Impact

Gas Savings.

Recommendation

Keep the intermediate vault validation in `_deployVault()` as the shared enforcement point and remove the duplicate check from each of the three creation functions.

Developer Response

Fixed in PR #294

2.9 Informational Findings

2.9.1 Update the `_checkLiqLTV()` initialization dependency comment

Technical Details

`__CollateralVaultBase_init()` initializes common state and then invokes `_checkLiqLTV()`. That validation obtains VaultManager parameters through `__targetAsset()` and reads the external protocol liquidation LTV through `_getExtLiqLTV()`. Both are virtual hooks whose implementations depend on concrete-vault state in addition to the common `asset`, `intermediateVault`, and `targetVault` values.

The comment currently located immediately before the validation reads:

```
1 // checkLiqLTV must happen after targetVault() and asset return meaningful values
```

For example, Aave uses `targetAsset`, `underlyingAsset`, and `categoryId`, while Morpho uses `targetAsset` and `mo_lltv`. Euler uses its immutable `targetAsset` and the common `asset` and `targetVault` values. The current concrete initializers establish these dependencies before calling the base initializer, so this is a documentation issue rather than a behavior defect.

Impact

Informational.

Recommendation

Update the comment to state that all fields read by the concrete implementations of `__targetAsset()` and `_getExtLiqLTV()` must be initialized before `_checkLiqLTV()` executes. Preserve this ordering in every concrete vault initializer.

Developer Response

Fixed in PR #299.

2.9.2 `withdraw()` can use external visibility

`CollateralVaultBase.withdraw()` is declared `public`, although production contracts only reach it through external calls. Using `external` would align the function visibility with its actual usage and may reduce generated bytecode.

Technical Details

Neither `CollateralVaultBase` nor any of its derived vault implementations calls `withdraw()` internally. The production call site in `MorphoDeleverageOperator` encodes `CollateralVaultBase.withdraw()` into an EVC batch, which invokes the function externally.

The `public` visibility therefore exposes an unused internal entry point. Declaring the function `external` preserves its current ABI and all existing external call paths while expressing the intended interface more precisely.

Impact

Informational.

Recommendation

Change the visibility of `CollateralVaultBase.withdraw()` from `public` to `external`.

Developer Response

Fixed in PR #294

yAudit

Twyne - Morpho Vault

Completed 2026-08-19