certora

Security Assessment Report



ORBT

February 2026

Prepared for ORBT Protocol

Table of Contents

Project Summary **3**
- Project Scope 3
- Project Overview 3
- Protocol Overview 4

Assessment Methodology **6**

Threat and Security Overview **7**
- Findings Summary 8
- Severity Matrix 8

Detailed Findings **9**
- High Severity Issues 11
  - H-01: StakingRewards governance integration is architecturally broken 11
  - H-02: ZeroXAsset cannot be paused as intended 14
- Medium Severity Issues 15
  - M-01: Repayments Could Be DoSed If Asset Reserves Are Frozen 15
  - M-02: swapExactOut() Would Not Work With Fee-On-Transfer Tokens 17
  - M-03: sZeroXAsset transfer whitelist check is not gated by isBeta which causes temporary restriction of transfers 19
  - M-04: Daily caps can be bypassed 21
  - M-05: Emergency pause mechanism fails due to immediate revert 24
- Low Severity Issues 26
  - L-01: perTxMaxUnderlying Can Be Bypassed Using A Multicall 26
  - L-02: IERC20.permit() calls can be DoSed by front-running 27
  - L-03: maxDeposit and maxMint return 0 for non-whitelisted users regardless of beta status 29
  - L-04: Pocket Can Steal All The pendingBuybackByAsset[] 31
  - L-05: Pockets Can Steal Other Pocket's Assets 32
  - L-06: feeUnderlying Should Be Rounded Up 33
- Informational Severity Issues 34
  - I-01: StakingRewards missing _disableInitializers() 34
  - I-02: Governance strategy registration bypasses validation 35
  - I-03: Any pocket can call privileged onlyOwner functions on the strategy, including upgradeTo 36
  - I-04: Pocket Can Avoid Fee By Directly Withdrawing From AAVE 37
  - I-05: Redundant check in OrbtUCE.sol::_reduceAllocatorDebt() 38
  - I-06: dailyCap Could Be Assigned A Value Lower Than mintedToday 39

Disclaimer **41**

About Certora **41**

certora ORBT

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
ORBT Protocol	https://github.com/Orbtofficial/uce https://github.com/Orbtofficial/usm https://github.com/Orbtofficial/rewards https://github.com/Orbtofficial/orbt-governance https://github.com/Orbtofficial/orbt-token https://github.com/Orbtofficial/orbt-mm https://github.com/Orbtofficial/strategy-core https://github.com/Orbtofficial/upm	Branch: 'main' in all repos	Solidity

Project Overview

This document describes the security review of **ORBT**. The work was undertaken from **February 2nd, 2026** to **February 20th, 2026**.

The following contract list is included in our scope:

```
Orbtofficial/uce/src/*
Orbtofficial/usm/src/*
Orbtofficial/rewards/src/*
Orbtofficial/upm/src/*
Orbtofficial/strategy-core/src/*
Orbtofficial/orbt-governance/src/*
Orbtofficial/orbt-token/src/*
Orbtofficial/orbt-mm/src/*
```

The team performed a manual audit of all the files in scope**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Protocol Overview

ORBT is a multi-module DeFi protocol that unifies fragmented collateral (WBTC, cbBTC, tBTC, etc.) into synthetic ZeroX tokens with 1:1 backing. Core modules:

- **UCE** — The Unified Conversion Engine (UCE) is the core routing and liquidity layer of the ORBT protocol, designed to unify fragmented Bitcoin (and other family) assets like WBTC, cbBTC, and tBTC into a single synthetic token (ZeroXBTC) that's always fully backed 1:1 by real underlying assets held in the contract. It handles seamless swaps between underlying assets and ZeroX, as well as between ZeroX and its yield-bearing vault form (sZeroX). Allocators can borrow ZeroX liquidity from the protocol based on their assigned credit lines, minted ZeroX stays locked in the contract until a real user swaps in equivalent underlying collateral — so the peg is never at risk. On top of that, the UCE handles oracle-based pricing except for base underlying asset which is always redeemable at a 1:1 rate.

- **ZeroXAsset** (ZeroXBTC, ZeroXETH, ZeroXUSD) — mintable ERC20 synthetics with permit, UUPS upgradeability, and pausable transfers. ZeroXAsset is always 1:1 with the base underlying asset of the family.

- **sZeroXAsset** (USM) — ERC-4626 vaults where ZeroXasset can be deposited with per-second compounding yield and a secondary reward stream from a pre-funded vault.

- **ORBT Governance** — M-of-N multisig with EIP-712 signatures, timelock, and digest-based replay protection.

- **ORBT Token** — 10B supply governance token with role-gated minting and UUPS upgradeability.

- **StakingRewards** —The ORBT Staking Rewards module lets ORBT token holders stake their tokens to earn rewards , it is a Synthetix-style staking with time-weighted reward distribution. The staking process is simple , stake ORBT, earn rewards proportional to your share of the total staked pool and how long you've been in, and withdraw whenever you want. Rewards are emitted at a constant rate over configurable periods (defaulting to 7 days)..

- **Strategy Core / UPM** — stateless adapters deploying collateral to Aave V3, with fee-on-profit and buyback-and-burn. UPM routes pocket calls to whitelisted strategies.

- **AcrossUCEBridge** — The ORBT Bridge module uses Across Protocol to move assets cross-chain without ever bridging ZeroX itself , only underlying assets (like USDC or WBTC)

actually travel. On the source chain, the adapter pulls the user's funds and deposits them into Across with bundled destination instructions. On arrival, a multicall handler feeds the underlying into the UCE, which mints ZeroX directly to the user. ZeroX-to-ZeroX flows work as a two-step redeem-then-mint, keeping supply clean and fully collateralized throughout. Per-tx caps, oracle freshness checks, and a reserve policy for incoming underlyings round out the safety model.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

Threat and Security Overview

Roles: ADMIN (pause, upgrades, whitelist), Governance (economic parameters via timelock), Allocators (credit lines, strategy deployment), Pockets (UPM-gated strategy calls), Users (swaps, yield, staking).

Key Invariants:

1. Circulating ZeroX is always 1:1 backed — allocator-borrowed ZeroX remains reserved in UCE.
2. Per-allocator debt bounded by ceiling and daily cap.
3. `exchangeRateRay` in sZeroXAsset can only increase (rateRay >= RAY).

Attack Vectors Considered:

- Oracle manipulation — feed staleness, price deviation bounds
- Allocator abuse — credit exhaustion, referral griefing, swap restriction bypass
- Reentrancy — nonReentrant across all modules, SafeERC20 for callbacks
- Governance — signature replay, signer compromise, unauthorized parameter changes
- ERC-4626 — first-depositor inflation, donation attacks (mitigated by stored exchange rate)
- Reward manipulation — transfer-to-self double-counting, retroactive claims
- Cross-chain bridge — daily cap bypass, unauthorized routes
- Privilege escalation — UPM doCall forwarding to strategy admin functions

Review Scope

Manual review of all Solidity source contracts across eight modules (~50 files). Focus areas: UCE swap/oracle/allocator logic, sZeroXAsset ERC-4626 compliance and reward accounting, governance signature/timelock/action routing, and strategy/UPM privilege boundaries.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	2	2	2
Medium	5	5	5
Low	6	6	4
Informational	6	6	3
Total	**19**	**19**	**14**

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

certora ORBT

Detailed Findings

ID	Title	Severity	Status
H-01	StakingRewards governance integration is architecturally broken	High	Fixed
H-02	ZeroXAsset cannot be paused as intended	High	Fixed
M-01	Repayments Could Be DoSed If Asset Reserves Are Frozen	Medium	Fixed
M-02	swapExactOut() Would Not Work With Fee-On-Transfer Tokens	Medium	Fixed
M-03	sZeroXAsset transfer whitelist check is not gated by isBeta which causes temporary restriction of transfers	Medium	Fixed
M-04	Daily caps can be bypassed	Medium	Fixed
M-05	Emergency pause mechanism fails due to immediate revert	Medium	Fixed
L-01	perTxMaxUnderlying Can Be Bypassed Using A Multicall	Low	Acknowledged
L-02	IERC20.permit() calls can be DoSed by front-running	Low	Acknowledged

L-03	maxDeposit and maxMint return 0 for non-whitelisted users regardless of beta status	Low	Fixed
L-04	Pocket Can Steal All The pendingBuybackByAsset[]	Low	Fixed
L-05	Pockets Can Steal Other Pocket's Assets	Low	Fixed
L-06	feeUnderlying Should Be Rounded Up	Low	Fixed
I-01	StakingRewards missing _disableInitializers()	Informational	Fixed
I-02	Governance strategy registration bypasses validation	Informational	Acknowledged
I-03	Any pocket can call privileged onlyOwner functions on the strategy, including upgradeTo	Informational	Fixed
I-04	Pocket Can Avoid Fee By Directly Withdrawing From AAVE	Informational	Acknowledged
I-05	Redundant check in OrbtUCE.sol::_reduceAllocatorDebt()	Informational	Fixed
I-06	dailyCap Could Be Assigned A Value Lower Than mintedToday	Informational	Acknowledged

High Severity Issues

H-01: StakingRewards governance integration is architecturally broken

Severity: High	Impact: Medium	Likelihood: High
Files: rewards/src/StakingRewards.sol	Status: Fixed	

Description:

The StakingRewards contract inherits GovernanceIntegration, which gates both _authorizeUpgrade and setGovernance with onlyGovernance:

```
function _authorizeUpgrade(address) internal override onlyGovernance {}

function setGovernance(address newGovernance) external onlyGovernance {
    governance = IORBTGovernance(newGovernance);
}
```

However, ORBTGovernance can only interact with target contracts through a single interface method:

```
// ORBTGovernance.sol
success = IGovernedContract(targetContract).executeGovernanceAction(actionType,
payload);
```

The governance module has no mechanism to make arbitrary external calls. It cannot call upgradeToAndCall() or setGovernance() on StakingRewards. Furthermore, StakingRewards._handleGovernanceAction() has no action type for either operation - only PAUSE, UNPAUSE, SET_REWARDS_DURATION, SET_REWARDS_DISTRIBUTION, SET_MAX_REWARD_RATE, and NOTIFY_REWARD_AMOUNT.

This creates two permanent deadlocks:

1. Upgrade doesn't work: The upgrade authorization requires a caller (`ORBTGovernance`) that structurally cannot make the required call (`upgradeToAndCall`). The contract is permanently non-upgradeable.

1. Governance migration doesn't work: If the governance contract is ever upgraded or migrated to a new address, StakingRewards cannot be updated to point to it. All governance-controlled parameters (reward rate, duration, reward token, notifications) become permanently frozen.

Note: This issue does NOT affect other upgradeable contracts:

- `OrbtUCE` and `sZeroXAsset` use `onlyRole(ADMIN)` — the admin can call `upgradeToAndCall` directly
- `BaseStrategy` uses `onlyOwner` where `owner = UPM` — UPM's `doCall` can invoke `upgradeToAndCall`

Recommendations:

Add UPGRADE and SET_GOVERNANCE action types to _handleGovernanceAction:

```
bytes32 private constant ACTION_UPGRADE = keccak256("UPGRADE");
bytes32 private constant ACTION_SET_GOVERNANCE = keccak256("SET_GOVERNANCE");

// In _handleGovernanceAction:
if (actionType == ACTION_UPGRADE) {
    (address newImpl, bytes memory data) = abi.decode(payload, (address, bytes));
    upgradeToAndCall(newImpl, data);
    return true;
} else if (actionType == ACTION_SET_GOVERNANCE) {
    address newGov = abi.decode(payload, (address));
    if (newGov == address(0)) revert ZeroGovernance();
    emit GovernanceSet(address(governance), newGov);
    governance = IORBTGovernance(newGov);
    return true;
}
```

Customer Response:

Fixed in 521ee85.

Fix Review:

Confirmed.

H-02: ZeroXAsset cannot be paused as intended

Severity: High	Impact: High	Likelihood: Medium
Files: src/token/base/ZeroXAsset.sol#L67	Status: Fixed	

Description:

The ZeroXAsset contract inherits PausableUpgradeable and exposes pause() and unpause() functions, but never enforces the pause state on transfers. According to the documentation, when paused, transfer and transferFrom should revert. However, the contract lacks the required _update() override to check the pause state.

As a result, calling pause() has no effect. Users can continue transferring tokens freely during what should be an emergency pause, completely defeating the purpose of the intended mechanism.

Recommendations:

Override _update to enforce the pause state on transfers while allowing mint/burn to continue (as per documented design).

Customer Response:

Fixed in a3eb3d3.

Fix Review:

Confirmed.

Medium Severity Issues

M-01: Repayments Could Be DoSed If Asset Reserves Are Frozen

Severity: Medium	Impact: Medium	Likelihood: Low
Files: strategy-core/src/OrbtMMStrategy	Status: Fixed	

Description:

Upon repayment in the OrbtMMStrategy a fee is calculated on top of the amount being repaid and a portion of this calculated fee is deposited in the AAVE pool →

```
if (feeCompound > 0) {

                asset.forceApprove(lendingPool, feeCompound);

                ILendingPool(lendingPool).deposit(address(asset), feeCompound,
pocket, 0);

                asset.forceApprove(lendingPool, 0);

                _increasePrincipal(aToken, pocket, feeCompound);

            }
```

Upon deposit in the AAVE pool the supply is validated through validateSupply() which requires the asset's reserve configuration to not be frozen →

```
(bool isActive, bool isFrozen, , , bool isPaused) = reserveCache

      .reserveConfiguration

      .getFlags();
```

```
    require(isActive, Errors.RESERVE_INACTIVE);

    require(!isPaused, Errors.RESERVE_PAUSED);

    require(!isFrozen, Errors.RESERVE_FROZEN);
```

The isFrozen flag is not checked on repayments and withdrawals , therefore even when repayments should be working smoothly the repay mechanism in the OrbtMMStrategy could be unnecessarily DoSed due to revert in deposit() of feeCompound.

Recommendations:

Wrap the deposit() call inside a try catch and inside the catch transfer the fees to treasury or the admin.

Customer Response:

Fixed in 5c1cf2a.

Fix Review:

Confirmed.

M-02: swapExactOut() Would Not Work With Fee-On-Transfer Tokens

Severity: Medium	Impact: Medium	Likelihood: Low
Files: uce/src/OrbtUCE	Status: Fixed	

Description:

The UCE module is supposed to be compatible with fee-on-transfer tokens but in the swapExactOut() logic →

```
// Deliver exact underlying out

            uint256 deliveredAmount = _settleUnderlyingOutbound(assetOut,
receiver, amountOut, referralCode, callerIsAllocator);

            if (deliveredAmount != amountOut) revert
Errors.OrbtUCE__SettlementMismatch();
```

Where amountOut is the exact output amount of token requested and delivered amount is calculated as →

```
function _settleUnderlyingOutbound(address assetOut, address receiver, uint256
amount, uint256 referralCode, bool callerIsAllocator) internal returns (uint256
delivered) {

        uint256 balanceBefore = IERC20(assetOut).balanceOf(receiver);

        address referralAllocator = referralToAllocator[referralCode];

        address allocatorForPocket = address(0);

        if (referralAllocator != address(0) && isAllocator(referralAllocator)) {

            allocatorForPocket = referralAllocator;

        } else if (callerIsAllocator) {
```

```
            allocatorForPocket = msg.sender;
        }
        _pushUnderlying(assetOut, receiver, amount, allocatorForPocket);
        delivered = IERC20(assetOut).balanceOf(receiver) - balanceBefore;
    }
```

The logic calculates how much was actually transferred to the receiver and for fee-on-transfer tokens the amount actually transferred would be less than “amount”.

Due to this the statement

```
if (deliveredAmount != amountOut) revert Errors.OrbtUCE__SettlementMismatch();
```

would revert even for valid token transfers.

Recommendations:

Compare the actual transferred amount without accounting for transfer fee.

Customer Response:

Fixed in 725b1d9.

Fix Review:

Confirmed.

M-03: sZeroXAsset transfer whitelist check is not gated by isBeta which causes temporary restriction of transfers

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: usm/src/sZeroXAsset.sol	Status: Fixed	

Description:

The _update function always enforces the BETA_LAUNCH_WHITELISTED role check on transfers, regardless of the isBeta flag:

```
function _update(address from, address to, uint256 value) internal override {
    _accrueRewards();
    if (to != address(0) && from != address(0)) {
        if (!hasRole(BETA_LAUNCH_WHITELISTED, to)) {
            revert("sZeroXAsset/receiver-not-whitelisted");
        }
    }
    // ...
}
```

The checkBeta modifier on deposit/withdraw/mint/redeem/claim properly checks if isBeta before enforcing the whitelist. But _update does not.

After beta is disabled, users can still deposit/withdraw/mint/redeem/claim, but they cannot transfer to non-whitelisted addresses.

This creates an operational problem. For the token to function post-beta, the admin must grant BETA_LAUNCH_WHITELISTED to every potential recipient (potentially thousands of users and contracts). It also contradicts the README: "Is sZeroXBTC transferable? Yes, fully transferable like any ERC-20."

Recommendations:

Wrap the whitelist check:

```
if (isBeta && to != address(0) && from != address(0)) {
    if (!hasRole(BETA_LAUNCH_WHITELISTED, to)) {
        revert("sZeroXAsset/receiver-not-whitelisted");
    }
}
```

If post-beta transfer restrictions are needed for compliance, consider implementing a separate blacklist mechanism.

Customer Response:

Fixed in 553a909.

Fix Review:

Confirmed.

M-04: Daily caps can be bypassed

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: uce/src/AcrossUCEBridge.sol	Status: Fixed	

Description:

The updateCaps() and setRoute() functions in AcrossUCEBridge unconditionally reset dayUsedUnderlying to 0 when called:

```
    function setRoute(
        ...
    ) external onlyOwner {
        //...
        // Initialize default caps for this destination
        TokenCaps storage caps =
capsByDestAndToken[destinationChainId][address(0)];
        caps.perTxMaxUnderlying = perTxMaxUnderlying_;
        caps.dailyMaxUnderlying = dailyMaxUnderlying_;
        caps.dayIndex = _currentDayIndex();
@>      caps.dayUsedUnderlying = 0;
    }

    function updateCaps(
        uint256 destinationChainId,
        address sourceToken,
        uint256 perTxMaxUnderlying_,
        uint256 dailyMaxUnderlying_
    ) external onlyOwner {
        RouteConfig storage route = routes[destinationChainId];
        if (!route.exists) revert RouteMissing(destinationChainId);

        TokenCaps storage caps =
capsByDestAndToken[destinationChainId][sourceToken];
        caps.perTxMaxUnderlying = perTxMaxUnderlying_;
        caps.dailyMaxUnderlying = dailyMaxUnderlying_;
```

```
        caps.dayIndex = _currentDayIndex();
@>      caps.dayUsedUnderlying = 0;
    }
```

The contract already has a natural daily reset mechanism in `_enforceCaps()` that correctly resets usage when a new UTC day begins:

```
    uint256 today = _currentDayIndex();
    if (caps.dayIndex != today) {
        caps.dayIndex = today;
        caps.dayUsedUnderlying = 0;
    }
```

However, any call to updateCaps() or setRoute() during the same day bypasses this mechanism by manually zeroing the counter.

If a user is aware of a pending admin config update, they can bridge up to the daily limit before the update and then bridge the full limit again after, effectively doubling the daily cap.

Example:

1. With a 1M USDC daily cap, a user bridges 1M USDC.
1. Admin calls `updateCaps()`.
1. `dayUsedUnderlying` resets to 0.
1. User bridges another 1M.

Total bridged: 2M — 2x the intended daily limit.

Recommendations:

Do not reset dayUsedUnderlying or dayIndex when updating caps. Let _enforceCaps() handle day transitions naturally:

For setRoute(), only initialize day tracking for new routes (when `caps.dayIndex == 0`), not when overwriting existing ones.

Customer Response:

Fixed in b037f42.

Fix Review:

Confirmed.

M-05: Emergency pause mechanism fails due to immediate revert

Severity: Medium	Impact: Low	Likelihood: High
Files: uce/src/OrbtUCE.sol#1354	Status: Fixed	

Description:

The _enforcePriceBounds function attempts to pause the contract when asset prices deviate beyond acceptable thresholds. However, the implementation immediately reverts after calling _pause(), which rolls back the pause state change along with the entire transaction.

```
function _enforcePriceBounds(address asset) internal {
    uint256 px = _priceAssetInBase1e18(asset);
    uint256 allowedDelta = uint256(priceDeviationBps) * 1e14;
    uint256 dist;
    unchecked {
        dist = px > RATE_PRECISION ? px - RATE_PRECISION : RATE_PRECISION - px;
    }
    if (dist > allowedDelta) {
        if (!paused()) _pause();
@>      revert Errors.OrbtUCE__AssetIsPaused();
    }
}
```

Recommendations:

Revert the operation instead of pausing in case a price deviation is detected:

```
if (dist > allowedDelta) {
-   if (!paused()) _pause();
-   revert Errors.OrbtUCE__AssetIsPaused();
+       revert Errors.OrbtUCE__ErrorMsg();
+   }
}
```

Customer Response:

Fixed in 3ed6569.

Fix Review:

Confirmed.

Low Severity Issues

L-01: perTxMaxUnderlying Can Be Bypassed Using A Multicall

Severity: Low	Impact: Low	Likelihood: Medium
Files: uce/src/AcrossUCEBridge	Status: Acknowledged	

Description:

The caps.perTxMaxUnderlying is supposed to be the per tx cap for a token when being bridged. A whitelisted user can make use of a multicall, within the multicall there can be multiple bridge tx batched(i.e. bridgeUnderlyingToChain() or bridgeZeroXToChain()) and since the multicall tx operates as one tx the total amount bridged would go beyond the perTxMaxUnderlying.

For example, if the perTxMaxUnderlying was 100 then a whitelisted user can batch 5 bridge calls in a multicall where all of them would be bridging 100 tokens, but since multicall operates as one tx the entire multicall tx would bridge 500 tokens in a single multicall tx.

Recommendations:

Consider having a stricter check like a per-block requirement for bridging else consider acknowledging the risk.

Customer Response:

Acknowledged.

L-02: IERC20.permit() calls can be DoSed by front-running

Severity: Low	Impact: Low	Likelihood: Low
Files: orbt-v3-core/contracts/protocol/pool/Pool.sol strategy-core/src/OrbtMMStrategy.sol uce/src/AcrossUCEBridge.sol uce/src/wrappers/IntegratorUCERouter.sol	Status: Acknowledged	

Description:

Six instances across the codebase call permit() without wrapping in a try-catch block:

```
IntegratorUCERouter.sol    - swapExactInWithPermit()
IntegratorUCERouter.sol    - swapExactOutWithPermit()
AcrossUCEBridge.sol        - bridgeUnderlyingToChainWithPermit()
OrbtMMStrategy.sol         - depositWithPermit()
Pool.sol                   - supplyWithPermit()
Pool.sol                   - repayWithPermit()
```

An attacker can extract the permit signature from a pending tx and front-run it by calling permit() directly on the token contract. Since the nonce is consumed by the attacker's transaction, the original user's transaction reverts. The impact is limited to a short-term DoS since separate non-permit paths exist for each function, allowing users to retry with standard approvals.

Notably, OpenZeppelin's own IERC20Permit interface explicitly recommends the try-catch pattern:

```
 * because permits have
 * built-in replay protection and can be submitted by anyone, they can be
frontrun. A protocol that uses permits should
 * take this into consideration and allow a `permit` call to fail. Combining
these two aspects, a pattern that may be
```

```
 * generally recommended is:
 *
 * ```solidity
 * function doThingWithPermit(..., uint256 value, uint256 deadline, uint8 v,
bytes32 r, bytes32 s) public {
 *     try token.permit(msg.sender, address(this), value, deadline, v, r, s) {}
catch {}
 *     doThing(..., value);
 * }
```

Recommendations:

Wrap all permit() calls in a try-catch block so the function proceeds regardless of whether the permit succeeds:

```
try IERC20Permit(token).permit(
    msg.sender, address(this), value, deadline, v, r, s
) {} catch {}
```

Customer Response:

Acknowledged.

L-03: maxDeposit and maxMint return 0 for non-whitelisted users regardless of beta status

Severity: Low	Impact: Low	Likelihood: Low
Files: usm/src/sZeroXAsset.sol	Status: Fixed	

Description:

maxDeposit and maxMint unconditionally check the BETA_LAUNCH_WHITELISTED role without consulting the isBeta flag:

```
function maxDeposit(address receiver) public view returns (uint256) {
    if (paused() || !hasRole(BETA_LAUNCH_WHITELISTED, receiver)) return 0;
    return type(uint256).max;
}

function maxMint(address receiver) public view returns (uint256) {
    if (paused() || !hasRole(BETA_LAUNCH_WHITELISTED, receiver)) return 0;
    return type(uint256).max;
}
```

Once beta is disabled, the checkBeta modifier on deposit() and mint() will allow any user to deposit. However, maxDeposit and `maxMint` still return 0 for non-whitelisted addresses.

Per ERC-4626, maxDeposit "MUST return the maximum amount of assets deposit would allow to be deposited for the receiver, through a deposit call." Returning 0 when the deposit would actually succeed violates this requirement.

Recommendations:

Gate the whitelist check on isBeta, consistent with checkBeta:

```
function maxDeposit(address receiver) public view returns (uint256) {
    if (paused()) return 0;
    if (isBeta && !hasRole(BETA_LAUNCH_WHITELISTED, receiver)) return 0;
```

```
    return type(uint256).max;
}
```

Customer Response:

Fixed in 33cbf6e.

Fix Review:

Confirmed.

L-04: Pocket Can Steal All The pendingBuybackByAsset[]

Severity: Low	Impact: Medium	Likelihood: Low
Files: strategy-core/src/OrbtMMStrategy	Status: Fixed	

Description:

Pocket address can trigger the executeBuybackAndBurn() in the BaseStrategy through the UPM module, when pendingBuybackByAsset[] for say assetA was 100 and provide minAmountOut as 1. This triggers swap() in buybackSwapper with amountIn as 100 and minAmountOut as 1 , since the "data" field can also be set by the pocket therefore swapData.executor can be an address in pocket's control , a swap() is initiated on 1Inch , at 1Inch 100 tokens are first transferred to the desc.srcReceiver which is in pocket's control i.e. swapData.executor and then _execute initiates which returns a returnAmount and our malicious executor can return just 1 as returnAmount and the minAmountOut checks will pass since 1 is the set slippage , meaning the pocket ran away with all assets from the strategy and provided just 1 wei ORBT in return.

Recommendations:

Consider restricting the executeBuybackAndBurn() to be only executed by the admin.

Customer Response:

Fixed in 3070919.

Fix Review:

Confirmed.

L-05: Pockets Can Steal Other Pocket's Assets

Severity: Low	Impact: High	Likelihood: Low
Files: strategy-core/src/OrbtMMStrategy	Status: Fixed	

Description:

A pocket can route a call through UPM and trigger a withdrawFromPocket() call on behalf of another pocket by providing the victim's pocket address in the pocket field and route the underlying withdrawn assets to an address in their control , consider the following scenario →

1. Pocket2 needs to withdraw from AAVE pool and will call `withdrawFromPocket()` and would have approved the necessary aTokens to the OrbtMMStrategy.
2. Pocket1 can frontrun this call and call `withdrawFromPocket()` with pocket address as Pocket2's address and "to" address as an address in their control and the withdrawn assets after applying fee will be transferred to the "to" address.

Recommendations:

Consider not allowing a pocket to withdraw for another pocket.

Customer Response:

Fixed in 584e706.

Fix Review:

Confirmed.

L-06: feeUnderlying Should Be Rounded Up

Severity: Low	Impact: Low	Likelihood: Medium
Files:	Status: Fixed	

Description:

When calculating feeUnderlying in allocatorRepay() ->

```
uint256 feeUnderlying = (assets * bps) / PERCENTAGE_PRECISION;

        if (feeUnderlying > 0) {

            if (treasury == address(0)) revert Errors.OrbtUCE__NoTreasury();

            IERC20(asset).safeTransfer(treasury, feeUnderlying);

        }
```

The fee calculation is being rounded down against the protocol, this could lead to scenarios where the debt is repaid without paying any borrow fee (consider the decimals of the asset to be small , for example OUSD with 2 decimals).

Recommendations:

Consider rounding up the feeUnderlying calculation in favor of the protocol.

Customer Response:

Fixed in 54c9f03.

Fix Review:

Confirmed.

Informational Severity Issues

I-01: StakingRewards missing _disableInitializers()

Files:

rewards/src/StakingRewards

Description:

The StakingRewards implementation contract does not call _disableInitializers() in a constructor. As a result the implementation address can be initialized by anyone.

Recommendations:

Add a constructor to the implementation:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Customer Response:

Fixed in 486db3c.

Fix Review:

Confirmed.

I-02: Governance strategy registration bypasses validation

Files:

upm/src/OrbtUPM.sol

Description:

The _validateStrategy function is commented out in the governance path, meaning governance can register any address as a strategy:

```
function _registerStrategy(address strategyAddress) internal {
    // _validateStrategy(strategyAddress);
    allowedStrategy[strategyAddress] = true;
}
```

Recommendations:

Consider uncommenting the _validateStrategy() function.

Customer Response:

Acknowledged.

I-03: Any pocket can call privileged onlyOwner functions on the strategy, including upgradeTo

Files:

strategy-core/src/BaseStrategy.sol
upm/src/OrbtUPM.sol

Description:

BaseStrategy sets its owner to the UPM contract during initialization:

```
_transferOwnership(upm_); // owner set to UPM
```

This makes every onlyOwner function on the strategy callable by the UPM. Meanwhile, OrbtUPM.doCall() allows any address with the POCKET or ADMIN role to forward arbitrary calldata to any whitelisted strategy:

```
function doCall(address target, bytes memory data)
    external whenNotPaused nonReentrant onlyPocketOrAdmin
    returns (bytes memory result)
{
    require(allowedStrategy[target], "UPM/strategy-not-allowed");
    return target.functionCall(data);
}
```

Since doCall executes with msg.sender = UPM (the strategy's owner), any pocket can invoke all `onlyOwner` functions without restriction. There is no selector filtering or calldata validation.

Customer Response:

Fixed in ed7a69f.

Fix Review:

Confirmed.

I-04: Pocket Can Avoid Fee By Directly Withdrawing From AAVE

Files:

strategy-core/src/OrbtMMStrategy

Description:

Admin can supply a pocket's address to a suitable AAVE pool which would generate a good steady profit , when withdrawing from AAVE through the OrbtMMStrategy a fee is applied on the withdrawn amount and the net amount of underlying asset transferred to the destination address is withdrawn amount - fee. But the pocket can directly withdraw from AAVE pool and bypass this fee from the OrbitMMStrategy i.e. after the pocket's position has made a good profit it can withdraw from AAVE since the aTokens are held by the pocket and this way the pocket does not have to pay the withdrawal fee.

Customer Response:

Acknowledged.

I-05: Redundant check in OrbtUCE.sol::_reduceAllocatorDebt()

Files:

uce/src/OrbtUCE.sol#1126

Description:

In OrbtUCE.sol::_reduceAllocatorDebt(), the check at line 1126 can never evaluate to true:

```
function _reduceAllocatorDebt(address allocator, uint256 oxValue, address payer,
bool strict) internal {
    if (oxValue == 0) return;
    uint256 debtBefore = alloc[allocator].baseDebt;
    if (debtBefore == 0) return;
    if (strict && oxValue > debtBefore) revert ...;

    uint256 repayAmount = oxValue < debtBefore ? oxValue : debtBefore;

@>  if (repayAmount == 0) return;
    ...
}
```

Since repayAmount = min(oxValue, debtBefore) and both values are guaranteed to be non-zero before that, the result is always positive.

Recommendations:

Remove the redundant check to improve code clarity and save a small amount of gas.

Customer Response:

Fixed in b3dfa91.

Fix Review:

Confirmed.

I-06: dailyCap Could Be Assigned A Value Lower Than mintedToday

Files:

uce/src/OrbtUCE.sol

Description:

When assigning/updating the depositCap in the _applyAllocatorUpdate() the following logic is invoked ->

```
} else if (op == IOrbtUCE.AllocatorOperations.UPDATE_LINE) {

            if (init.line.ceiling == 0) revert Errors.OrbtUCE__NoCreditLine();

            if (init.line.dailyCap == 0) revert
Errors.OrbtUCE__DailyCapExceeded();

            if (allocatorDebt(init.allocator) > init.line.ceiling) revert
Errors.OrbtUCE__CeilingExceeded();

            // Preserve mintedToday/lastMintDay by only updating limits

            a.line.ceiling = init.line.ceiling;

            a.line.dailyCap = init.line.dailyCap;

        }
```

The dailyCap is assigned directly without checking if the new updated value is lower than the line.mintedToday.

An allocator can take advantage of this in the following example scenario ->

1.) Allocator's dailyCap is 100.

2.) Admin calls setAllocatorSingleByAdmin() to update the allocator's dailyCap to 70.

3.) Allocator frontruns with a allocatorCreditMint() and mint 100 total debt.

Therefore when the dailyCap is updated to 70 the lastMinted has gone beyond the dailyCap to 100.

Recommendations:

Consider checking if the updated dailyCap is higher than the line.lastMinted in _applyAllocatorUpdate

Customer Response:

Acknowledged.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.