hashlock.

Security Audit

dTRINITY (DeFi)

October 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

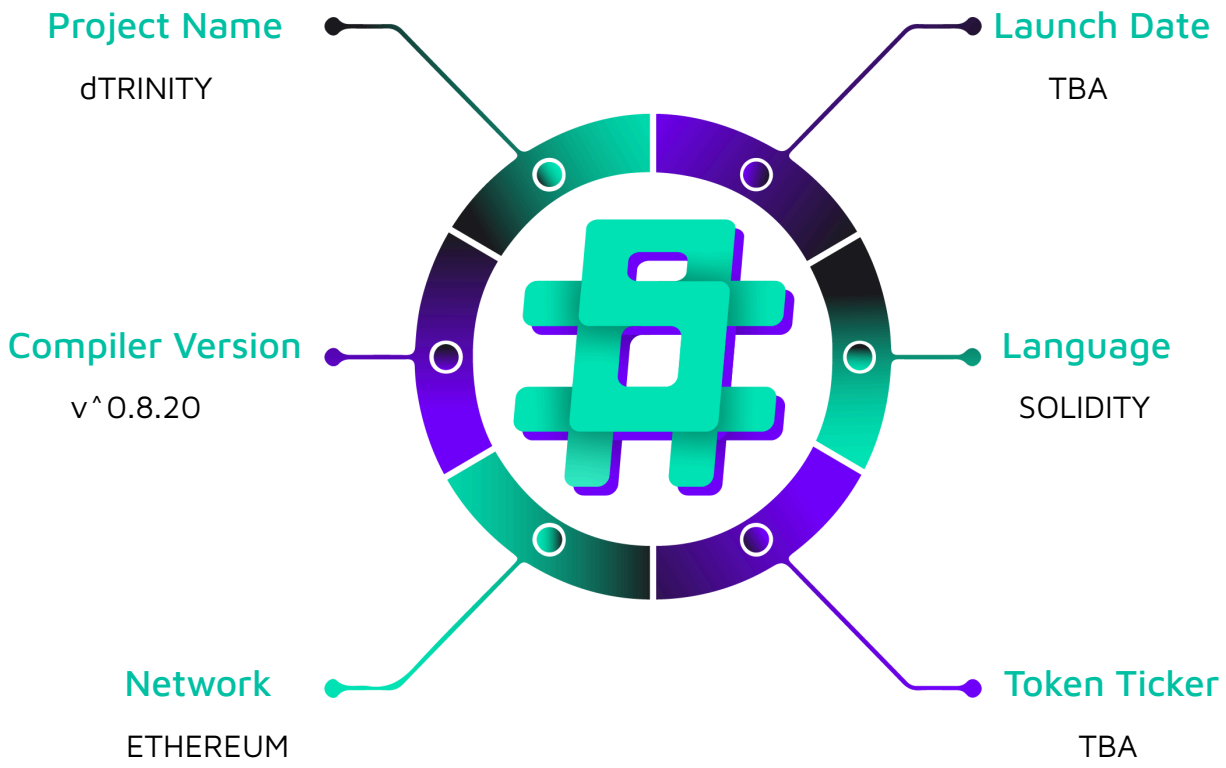
The dTRINITY team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

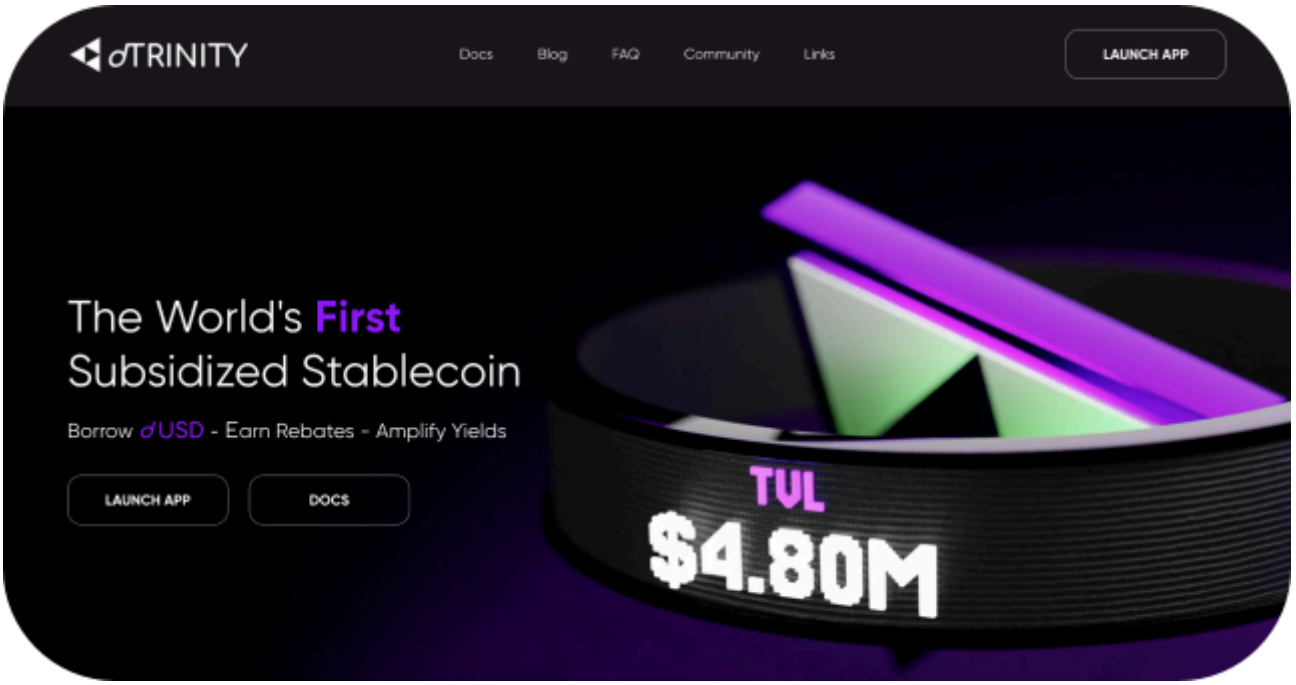
dTRINITY is a DeFi protocol that issues subsidized stablecoins—such as dUSD—and operates its own lending market designed to make on-chain borrowing cheaper while offering enhanced yields to lenders and liquidity providers. It uses yield-bearing stablecoin reserves to maintain a 1:1 peg and redirects the yield as "rebates" to borrowers, effectively lowering their borrowing costs. At the same time, lenders earn competitive interest rates and participation rewards. The protocol launched on the Fraxtal L2 in late 2024 and is expanding to additional networks, with a governance token planned after the TGE to empower community-driven decision-making.

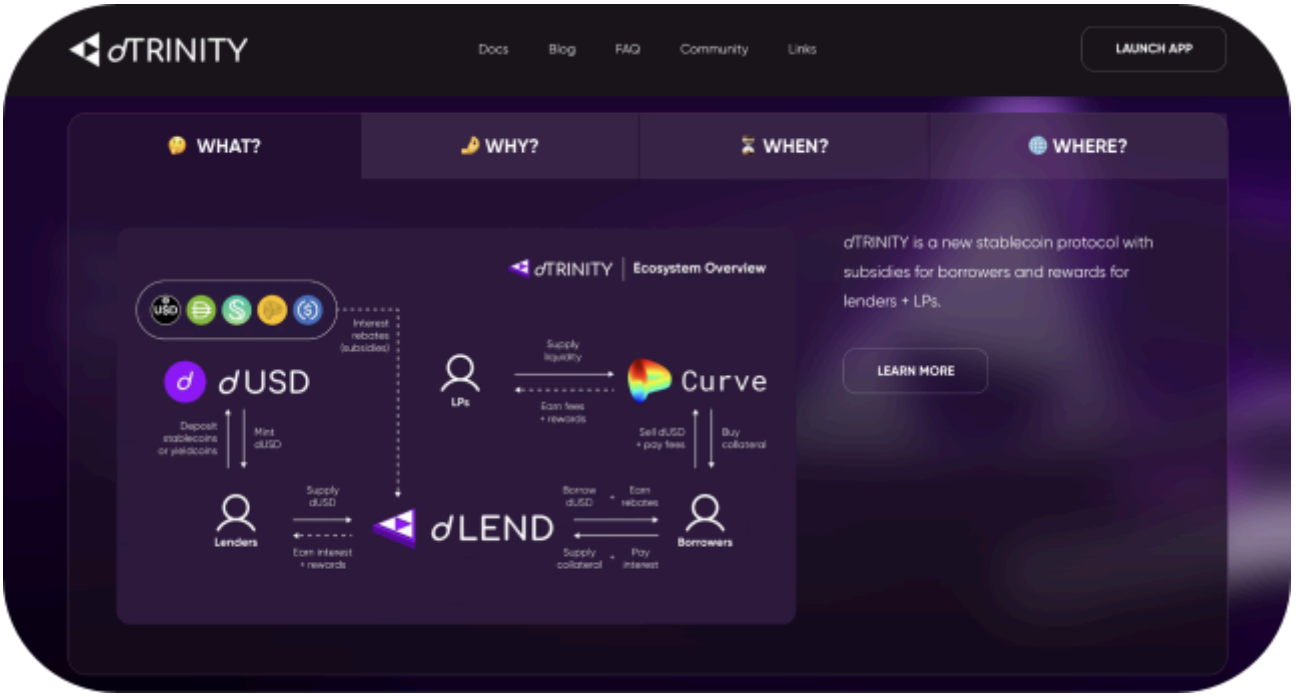
Project Name: dTRINITY
Project Type: DeFi
Compiler Version: ^0.8.20
Website: https://www.dtrinity.org/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the dTRINITY project. The scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**dTRINITY Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**October, 2025**
Contract 1	contracts/dstable/IssuerV2_1.sol
Contract 2	contracts/dstable/RedeemerV2.sol
Contract 3	contracts/dstable/AmoManagerV2.sol
Contract 4	contracts/dstable/AmoDebtToken.sol
Contract 5	contracts/dlend/periphery/adapters/odos/BaseOdosSwapAdapter.sol
Contract 6	contracts/dlend/periphery/adapters/odos/BaseOdosSellAdapterV2.sol
Contract 7	contracts/dlend/periphery/adapters/odos/BaseOdosBuyAdapterV2.sol
Contract 8	contracts/dlend/periphery/adapters/odos/OdosDebtSwapAdapterV2.sol
Contract 9	contracts/dlend/periphery/adapters/odos/OdosLiquiditySwapAdapterV2.sol

Contract 10	contracts/dlend/periphery/adapters/odos/OdosRepayAdapterV2.sol
Contract 11	contracts/dlend/periphery/adapters/odos/OdosWithdrawSwapAdapterV2.sol
Contract 12	contracts/dlend/periphery/adapters/odos/OracleValidation.sol
Contract 13	contracts/dlend/periphery/adapters/odos/PendleSwapLogic.sol
Contract 14	contracts/dlend/periphery/adapters/odos/SafeOracleMath.sol
Contract 15	contracts/dlend/periphery/adapters/odos/SwapExecutor.sol
Contract 16	contracts/odos/OdosSwapUtils.sol
Contract 17	contracts/common/SupportsWithdrawalFee.sol
Contract 18	contracts/vaults/rewards_claimable/RewardClaimable.sol
Contract 19	contracts/vaults/vesting/ERC20VestingNFT.sol
Contract 20	contracts/vaults/dstake/DStakeCollateralVaultV2.sol
Contract 21	contracts/vaults/dstake/DStakeRouterV2.sol
Contract 22	contracts/vaults/dstake/DStakeTokenV2.sol
Contract 23	contracts/vaults/dstake/adapters/GenericERC4626ConversionAdapter.sol
Contract 24	contracts/vaults/dstake/adapters/MetaMorphoConversionAdapter.sol

Contract 25	contracts/vaults/dstake/libraries/AllocationCalculator.sol
Contract 26	contracts/vaults/dstake/libraries/DeterministicVaultSelector.sol
Contract 27	contracts/vaults/dstake/rewards/DStakeRewardManagerMetaMorpho.sol
Contract 28	contracts/vaults/dstake/vaults/DStakeIdleVault.sol
Audited GitHub Commit Hash	5aebb584a8dd52e03131dbe1e8c3a17ca7837b4b
Fix Review GitHub Commit Hash	9a31268a8106acf9d7a7941a78237239a34dd4b2

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

5 High-severity vulnerabilities

4 Low-severity vulnerabilities

7 QA

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
IssuerV2_1.sol - Act as a minting controller for dStable tokens backed by collateral | Contract achieves this functionality. | - Allow users to deposit collateral and receive dStable via `issue(...)` with slippage protection - Manage minting through per-asset pause, global pause, and collateral backing verification - Provide role-based access control for admin functions - Validate zero addresses on parameters - Allow collateral vault updates via `setCollateralVault(...)` - Mint from excess backing via `issueUsingExcessCollateral(...)`	**Contract achieves this functionality.**
RedeemerV2.sol - Act as a redemption controller for burning dStable and returning collateral - Allow users to burn dStable and receive collateral via `redeem(...)` with slippage protection - Manage redemption through per-asset pause, global pause, and fee-free protocol redemption - Handle fees with default rate (up to 5% max) and per-collateral overrides - Handle fees with default rate (up to 5% max) and per-collateral overrides - Allow fee receiver updates via `setFeeReceiver(...)	**Contract achieves this functionality.**

AmoManagerV2.sol - Act as a unified AMO operations manager for stable and collateral AMO - Allow atomic increase/decrease of AMO supply via `increaseAmoSupply(...)` / `decreaseAmoSupply(...)` - Allow borrow/repay of collateral via `borrowTo(...)` / `repayFrom(...)` - Support EIP-2612 permit via `repayWithPermit(...)` - Enforce invariant checks with configurable tolerance - Prevent operations during off-peg conditions - Manage access via roles and allowlisted wallets	- Unconditional `permit()` call can be front-run, causing the transaction revert
AMODebtToken.sol - Act as an ERC20 receipt token for AMO operations with transfer restrictions - Allow AMO Manager to mint/burn debt tokens to/from allowlisted vaults - Enforce allowlist-based transfer control - Manage access via `AMO_MANAGER_ROLE` and `DEFAULT_ADMIN_ROLE`	**Contract achieves these functionalities correctly**
BaseOdosSwapAdapter.sol - Act as a base utility contract for Odos swap adapters - Pull aTokens and withdraw underlying via `_pullATokenAndWithdraw(...)` - Conditionally renew allowances via `_conditionalRenewAllowance(...)` - Provide rescue and pause admin functions - Validate constructor parameters	Unconditionally calls `permit()` when a signature is provided; front-runners can consume it, causing a revert.
BaseOdosSellAdapterV2.sol	Returns the input amount

- Act as a base adapter for adaptive selling with multi-protocol support - Act as a base adapter for adaptive selling with multi-protocol support - Execute direct Odos-only swaps via `_executeDirectOdosExactInput(...)` - Validate swaps against Oracle prices	spent instead of the output amount received
BaseOdosBuyAdapterV2.sol - Act as a base adapter for adaptive buying with multi-protocol support - Execute adaptive buy routing between Odos and Pendle - Execute adaptive buy routing between Odos and Pendle - Execute direct Odos-only swaps	For PT swaps, passes `maxInputAmount` directly, consuming the entire slippage buffer Returns the input amount spent instead of the output amount received
OdosDebtSwapAdapterV2.sol - Act as an adapter to swap debt to another debt type with PT token support - Swap debt via `swapDebt(...)` - Use credit delegation permits	`delegationWithSig()` called unconditionally; can be front-run. Nested flash-loan pulls `debtRepayAmount` instead of `extraCollateralAmount`; users lose multiples of authorized collateral
OdosLiquiditySwapAdapterV2.sol - Act as an adapter to swap liquidity using Odos with PT token support - Swap collateral to a different asset via `swapLiquidity(...)` - Handle permits for aToken withdrawals	Inherits permit front-running vulnerability. Inherits the wrong return value bug from OdosSwapUtils.
OdosRepayAdapterV2.sol	- Inherits permit

- Act as an adapter for repaying debt using a different asset as a source - Repay debt using collateral via `repayWithCollateral(...)` - Handle permits for aToken withdrawals - Handle permits for aToken withdrawals	front-running vulnerability - PT swaps consume full slippage buffer.
OdosWithdrawSwapAdapterV2.sol - Act as an adapter to withdraw and swap using Odos with PT token support - Withdraw aTokens and swap via `withdrawAndSwap(...)` - Handle permits for aToken withdrawals	**Contract achieves these functionalities correctly**
PendleSwapLogic.sol - Act as a library for handling PT token operations and composed swaps - Detect PT tokens and execute various swap paths - Execute composed PT → target swaps via `executePTToTargetSwap(...)`	Called with `maxInputAmount` as `ptAmount`, consuming entire slippage buffer
OracleValidation.sol - Act as shared oracle price validation logic for V2 adapters - Validate swaps against oracle prices with configurable tolerance (max 5%) - Reject zero-amount swaps and unconfigured oracle price	**Contract achieves these functionalities correctly**
SafeOracleMath.sol - Act as a library for safe arithmetic in Oracle calculations - Calculate expected amounts with overflow protection using `Math.mulDiv`	**Contract achieves these functionalities correctly**

SwapExecutor.sol - Act as a unified library for executing all types of swaps - Execute exact input swaps with automatic routing - Execute exact output swaps with automatic routing - Contains REGULAR_SWAP branch	Returns `actualAmountSpent` (input) instead of `actualAmountReceived` (output); all callers mis-account tokens. Credits user-controlled `exactOut` without verification; attackers can drain adapter balances.
SupportsWithdrawalFee.sol - Act as an abstract contract for vaults with withdrawal fee support - Initialize, set, and calculate withdrawal fees with maximum validation	**Contract achieves these functionalities correctly**
RewardClaimable.sol - Act as an abstract contract for vaults with a claimable reward - Compound rewards, set treasury address/fee, and exchange threshold - Provide reentrancy protection and parameter validation	**Contract achieves these functionalities correctly**
ERC20VestingNFT.sol - Act as a soft locker contract for dSTAKE tokens with a vesting period - Allow deposit, early redemption, and matured withdrawal with soul-bound status - Prevent transfer of mature NFTs and enforce deposit thresholds - Generate on-chain SVG metadata	**Contract achieves these functionalities correctly**

GenericERC4626ConversionAdapter.sol - Simple adapter that deposits/withdraws dSTABLE into a generic ERC-4626 vault and returns vault shares; used to bridge dSTABLE ↔ vault shares for the router.	**Contract achieves this functionality.**
MetaMorphoConversionAdapter.sol - Secure adapter for MetaMorpho ERC-4626 vaults with slippage checks, approvals management, and safety guards when converting dSTABLE to/from MetaMorpho shares.	**Contract achieves this functionality.**
AllocationCalculator.sol - Stateless helper to compute each vault's allocation in basis points from their balances (used to determine target weights).	**Contract achieves this functionality.**
DeterministicVaultSelector.sol - Picks the top under-/over-allocated vaults deterministically (partial sort) for rebalancing deposits/withdrawals.	**Contract achieves this functionality.**
DStakeRewardManagerMetaMorpho.sol - Claims and processes rewards from MetaMorpho (via URD), skims/compounds claimed dSTABLE back into collateral vault, and manages rewards distribution logic.	**Contract achieves this functionality.**

DStakeIdleVault.sol - Minimal ERC-4626 vault that holds idle dSTABLE and streams time-based emissions to depositors (handles reward reserve and vesting).	**Contract achieves this functionality.**
DStakeCollateralVaultV2.sol - Holds various strategy share tokens and converts their value into dSTABLE via registered adapters; acts as the collateral store for the DStake system and enforces router access.	**Contract achieves this functionality.**
DStakeTokenV2.sol - ERC-4626 wrapper (dSTAKE) representing shares of managed collateral; handles deposits/withdrawals, fees, router hooks, and accounting against total managed assets and shortfalls.	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the dTRINITY project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the dTRINITY project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] OdosSwapUtils#executeSwapOperation -Returns input amount instead of output amount, draining adapters

Description

The executeSwapOperation() helper calculates the output tokens received but returns the input amount spent. All V2 adapters that trust this return value mispay users and leak funds when dust is available on the adapter.

Vulnerability Details

The executeSwapOperation(...) function in OdosSwapUtils.sol records the destination token balance before the swap, executes the Odos router call, extracts actualAmountSpent from the return data, and calculates actualAmountReceived via balance delta. However, it incorrectly returns actualAmountSpent instead of actualAmountReceived:

```
function executeSwapOperation(
    IOdosRouterV2 router,
    address inputToken,
    address outputToken,
    uint256 maxIn,
    uint256 exactOut,
    bytes memory swapData
) internal returns (uint256 actualAmountSpent) {
    uint256 outputBalanceBefore = IERC20(outputToken).balanceOf(address(this));
    // ...
    assembly {
        // @audit extracts input amount spent from router return data
        actualAmountSpent := mload(add(result, 32))
    }
```

```
    uint256 outputBalanceAfter = IERC20(outputToken).balanceOf(address(this));
    uint256 actualAmountReceived;
    // ...
    actualAmountReceived = outputBalanceAfter - outputBalanceBefore;
    // ...
    if (actualAmountReceived < exactOut) {
        revert InsufficientOutput(exactOut, actualAmountReceived);
    }
    // ...
    // @audit BUG: Returns input amount instead of output amount
    return actualAmountSpent;
}
```

Multiple adapter contracts consume this return value. In `BaseOdosSellAdapterV2._executeDirectOdosExactInput(...)`:

```
function _executeDirectOdosExactInput(
    address inputToken,
    address outputToken,
    uint256 exactInputAmount,
    uint256 minOutputAmount,
    bytes memory swapData
) internal returns (uint256 actualOutputAmount) {
    // @audit Return value is named actualOutputAmount, but receives actualAmountSpent
    actualOutputAmount = OdosSwapUtils.executeSwapOperation(
        odosRouter,
        inputToken,
        outputToken,
        exactInputAmount,
        minOutputAmount,
        swapData
    );
    return actualOutputAmount;}
```

The incorrect value then flows to `OdosWithdrawSwapAdapterV2.withdrawAndSwap(...)`

```
function withdrawAndSwap(
    WithdrawSwapParamsV2 memory withdrawSwapParams,
    PermitInput memory permitInput
) external nonReentrant whenNotPaused {
    // ...
    // @audit amountReceived is actually the INPUT amount spent, not output received
    uint256 amountReceived = _executeAdaptiveSwap(
        IERC20Detailed(withdrawSwapParams.oldAsset),
        IERC20Detailed(withdrawSwapParams.newAsset),
        withdrawSwapParams.oldAssetAmount,
        withdrawSwapParams.minAmountToReceive,
        withdrawSwapParams.swapData
    );
    // ...
    // @audit Transfers INPUT amount to user, not OUTPUT amount
    IERC20(withdrawSwapParams.newAsset).safeTransfer(user, amountReceived);
}
```

Affected Code Paths:

1. `OdosWithdrawSwapAdapterV2.withdrawAndSwap(...)` → `_executeAdaptiveSwap(...)` → `_executeDirectOdosExactInput(...)` → `OdosSwapUtils.executeSwapOperation(...)`

2. `OdosLiquiditySwapAdapterV2._swapAndDeposit(...)` → `_executeAdaptiveSwap(...)` → `_executeDirectOdosExactInput(...)` → `OdosSwapUtils.executeSwapOperation(...)`

3. `OdosLiquiditySwapAdapterV2.executeOperation(...)` → `_executeAdaptiveSwap(...)` → `_executeDirectOdosExactInput(...)` → `OdosSwapUtils.executeSwapOperation(...)`

Proof of Concept

Scenario: Alice wants to swap 100 USDC for USDT using `withdrawAndSwap(...)`.
1. Alice calls `withdrawAndSwap(...)` with `oldAssetAmount = 100e6` (100 USDC) and `minAmountToReceive = 95e6` (95 USDT)
2. The adapter pulls 100 aUSDC from Alice and withdraws 100 USDC from Aave
3. The adapter calls `_executeAdaptiveSwap(...)` which routes to `_executeDirectOdosExactInput(...)`

4. `OdosSwapUtils.executeSwapOperation(...)` executes the swap:
   - Odos router consumes 100 USDC as input
   - Odos router produces 95 USDT as output
   - The router returns `100` (amount of input spent)
   - Function calculates `actualAmountReceived = 95` via balance delta
   - Function validates `95 >= 95` (passes)
   - BUG: Function returns `actualAmountSpent = 100` instead of `actualAmountReceived = 95`
5. Back in `withdrawAndSwap(...)`, the variable `amountReceived = 100`
6. The adapter attempts to transfer 100 USDT to Alice via `safeTransfer(user, amountReceived)`
7. Result:
   - If the adapter has 5+ USDT dust balance: Alice receives 100 USDT (5 USDT stolen from the adapter)
   - If the adapter has insufficient dust: Transaction reverts due to insufficient balance

Impact

All exact-input swap flows (`OdosWithdrawSwapAdapterV2`, `OdosCollateralSwapAdapterV2`, `OdosDebtSwapAdapterV2`) mis-account the received tokens. Swaps either revert (when dust is insufficient) or silently drain adapter balances whenever dust exists, creating a persistent and reproducible loss of funds.

Recommendation

Return `actualAmountReceived` instead of `actualAmountSpent` and review every caller that presently assumes the return value equals output tokens. Balance-delta-based accounting (used in V1) should be reintroduced to avoid similar mistakes.

```
function executeSwapOperation(
    // ...
) internal returns (uint256 actualAmountSpent) {
    // ...
    // Fixed: Return the output amount, not the input amount
    return actualAmountReceived;
}
```

Status

Resolved

[H-02] OdosDebtSwapAdapterV2#executeOperation - Pulls debtRepayAmount instead of user-authorized extraCollateralAmount

Description

When extraCollateralAsset is set, the nested flash-loan path pulls flashParams.debtRepayAmount aTokens from the user instead of the flash-loaned amount (which equals extraCollateralAmount). Users can lose multiples of the collateral they explicitly consented to supply.

Vulnerability Details

The swapDebt(...) function allows users to provide optional extra collateral via the extraCollateralAmount parameter. When this is set, the protocol initiates a flash loan for exactly extraCollateralAmount

```
function swapDebt(
    DebtSwapParamsV2 memory debtSwapParams,
    CreditDelegationInput memory creditDelegationPermit,
    PermitInput memory collateralATokenPermit
) external nonReentrant whenNotPaused {
    // ...
    if (debtSwapParams.extraCollateralAsset != address(0)) {
        // ...
        // @audit Flash loan initiated for extraCollateralAmount (user-authorized
amount)
        _flash(flashParams, debtSwapParams.extraCollateralAsset,
debtSwapParams.extraCollateralAmount);
    }
    // ...
}
```

Inside the executeOperation(...) callback, the flash-loaned amount equals extraCollateralAmount. However, the nested flash loan branch incorrectly pulls flashParams.debtRepayAmount:

```
function executeOperation(
    address[] memory assets,
    uint256[] memory amounts,
    uint256[] memory premiums,
    address initiator,
    bytes memory params
) external override returns (bool) {
    // ...
    uint256 amount = amounts[0];  // @audit This is extraCollateralAmount
    // ...
    FlashParamsV2 memory flashParams = abi.decode(params, (FlashParamsV2));


    if (flashParams.nestedFlashloanDebtAsset != address(0)) {
        (, , address aToken) = _getReserveData(asset);
        // @audit BUG: Pulls debtRepayAmount instead of amount (extraCollateralAmount)
        IERC20WithPermit(aToken).safeTransferFrom(flashParams.user, address(this),
flashParams.debtRepayAmount);
        POOL.withdraw(asset, flashParams.debtRepayAmount, address(this));
        // ...
    }
    // ...
}
```

Proof of Concept

Bob wants to swap his 5,000 USDC debt to DAI, providing 1,000 USDC as extra collateral for safety.

1. Bob calls `swapDebt(...)` with:
   - `debtRepayAmount = 5000e6` (5,000 USDC of old debt)
   - `extraCollateralAmount = 1000e6` (1,000 USDC authorised extra)
   - `extraCollateralAsset = USDC`
2. The adapter initiates a flash loan for `1000e6` USDC (the authorised `extraCollateralAmount`)
3. Inside `executeOperation(...)`:
   - `amount = 1000e6` (the flash-loaned extraCollateralAmount)

- flashParams.debtRepayAmount = 5000e6

4. BUG: The adapter executes:

```
IERC20WithPermit(aToken).safeTransferFrom(flashParams.user, address(this),
flashParams.debtRepayAmount);
```

This pulls 5,000 aUSDC from Bob instead of the authorised 1,000 aUSDC

5. Result: Bob loses 5,000 USDC of collateral when he only authorised 1,000 USDC

Impact

Every nested extra-collateral swap seizes the entire legacy debt value instead of the capped extra collateral, leading to unrecoverable theft of user funds. The vulnerability has 100% reproducibility whenever users use the extra collateral feature.

Recommendation

Transfer and withdraw `amount` (the flash-loaned value) instead of `flashParams.debtRepayAmount`:

```
if (flashParams.nestedFlashloanDebtAsset != address(0)) {
    (, , address aToken) = _getReserveData(asset);
    // Fixed: Use amount (extraCollateralAmount) not debtRepayAmount
    IERC20WithPermit(aToken).safeTransferFrom(flashParams.user, address(this), amount);
    POOL.withdraw(asset, amount, address(this));
    // ...
}
```

Status

Resolved

[H-03] SwapExecutorV2#executeSwapExactOutput -PT swaps spend the full maxInputAmount

Description

Exact-output PT swaps pass maxInputAmount straight into PendleSwapLogic as the amount to swap. No attempt is made to compute the minimum PT required, so user slippage buffers are always consumed and trapped on the adapter (Rescuable allows governance to sweep the surplus).

Vulnerability Details

In SwapExecutor.executeSwapExactOutput(...), every PT path forwards params.maxInputAmount to Pendle helper functions without computing the minimum required:

```
function executeSwapExactOutput(ExactOutputParams memory params) internal returns
(uint256 actualInputAmount) {
    ISwapTypes.SwapType swapType = PendleSwapLogic.determineSwapType(params.inputToken,
params.outputToken);
    // ...
    if (swapType == ISwapTypes.SwapType.PT_TO_REGULAR) {
        return
            // @audit maxInputAmount passed directly as ptAmount to swap
            PendleSwapLogic.executePTToTargetSwap(
                params.inputToken,
                params.outputToken,
                params.maxInputAmount,  // @audit This becomes ptAmount - entire max is
swapped
                params.exactOutputAmount,
                params.pendleRouter,
                params.odosRouter,
                ptSwapData
            );
    }
    // ...
}
```

The `exsecutePTToTargetSwap(...)` function treats this parameter as the exact amount to swap:

```
function executePTToTargetSwap(
    address ptToken,
    address targetToken,
    uint256 ptAmount,  // @audit This is maxInputAmount, treated as exact amount
    uint256 minTargetOut,
    address pendleRouter,
    IOdosRouterV2 odosRouter,
    PTSwapDataV2 memory swapData
) internal returns (uint256 actualTargetOut) {
    // ...
    // @audit Swaps the FULL ptAmount, not the minimum needed
    uint256 underlyingReceived = executePendleSwap(
        ptToken,
        ptAmount,  // @audit Entire maxInputAmount consumed
        swapData.underlyingAsset,
        pendleRouter,
        swapData.pendleCalldata
    );
    // ...
}
```

The `executePendleSwap(...)` function calls `PendleSwapUtils.swapExactInput(...)`:

```
function executePendleSwap(
    address ptToken,
    uint256 ptAmount,
    address underlyingAsset,
    address pendleRouter,
    bytes memory swapData
) internal returns (uint256 actualUnderlyingOut) {
    // ...
    // @audit Exact-input swap - spends entire ptAmount unconditionally
    PendleSwapUtils.swapExactInput(ptToken, underlyingAsset, ptAmount, pendleRouter,
swapData);
    // ...
```

```
}
```

Affected code paths

1. `OdosDebtSwapAdapterV2.executeOperation(...)` → `_executeAdaptiveBuy(...)` → `_executeComposedSwapExactOutput(...)` → `SwapExecutor.executeSwapExactOutput(...)` → `PendleSwapLogic.executePTToTargetSwap(...)`
2. `OdosRepayAdapterV2.repayWithCollateral(...)` → `_executeAdaptiveBuy(...)` → `_executeComposedSwapExactOutput(...)` → `SwapExecutor.executeSwapExactOutput(...)` → `PendleSwapLogic.executePTToTargetSwap(...)`

Proof Of Concept

Carol wants to repay exactly 900 USDC of debt using PT-USDC collateral with a 10% slippage buffer.

1. Carol calls `repayWithCollateral(...)` with:
   - `collateralAmount = 1000e6` (1,000 PT-USDC as max input, including 10% buffer)
   - `repayAmount = 900e6` (900 USDC debt to repay)
2. The adapter calls `_executeAdaptiveBuy(...)` with `maxAmountToSwap = 1000e6`
3. Due to favourable market prices, only 850 PT-USDC is actually needed to get 900 USDC
4. However, `SwapExecutor.executeSwapExactOutput(...)` passes `maxInputAmount = 1000e6` to Pendle
5. `executePTToTargetSwap(...)` swaps all 1,000 PT-USDC via `swapExactInput(...)`
6. The swap yields approximately 1,060 USDC (more than the 900 needed)
7. The adapter uses 900 USDC for debt repayment
8. 160 USDC remains trapped on the adapter contract
9. Governance can later call `rescueToken(...)` to sweep these funds

Impact

All PT exact-output flows (`OdosDebtSwapAdapterV2`, `OdosRepayAdapterV2`, etc.) siphon users' slippage buffers into adapter balances. Large positions (5-figure USD) lose thousands of dollars per execution, and governance can later rescue the trapped funds.

Recommendation

Estimate the required PT amount before executing, ensure it does not exceed `maxInputAmount`, trade only what is needed, and return unused collateral to the user:

```
function executePTToTargetSwap(
    address ptToken,
    address targetToken,
    uint256 maxPtAmount,  // Renamed for clarity
    uint256 exactTargetOut,
    // ...
) internal returns (uint256 actualTargetOut) {
    // Calculate minimum PT needed using oracle/SDK quote
    uint256 ptNeeded = _estimatePTForOutput(exactTargetOut, ptToken, targetToken);
    require(ptNeeded <= maxPtAmount, "Exceeds max input");

    // Swap only what's needed
    uint256 underlyingReceived = executePendleSwap(ptToken, ptNeeded, ...);
    // ...
}
```

Status

Resolved

[H-04] BaseOdosSwapAdapter#_pullATokenAndWithdraw & OdosDebtSwapAdapterV2#swapDebt -Permit/credit-delegation signatures can be front-run to DoS users

Description

The _pullATokenAndWithdraw(...) and swapDebt(...) unconditionally call permit()/delegationWithSig() whenever a signature struct is supplied. Any mempool observer can front-run those permit calls with the same signature, consume the nonce, and make the user transaction revert once it reaches the block.

Vulnerability Details

The _pullATokenAndWithdraw(...) function in BaseOdosSwapAdapter.sol calls permit() whenever permitInput.deadline != 0 without checking existing allowance or handling already-consumed signatures:

```
function _pullATokenAndWithdraw(
    address reserve,
    address user,
    uint256 amount,
    PermitInput memory permitInput
) internal returns (uint256) {
    if (permitInput.deadline != 0) {
        // @audit Unconditional permit call - vulnerable to front-running
        permitInput.aToken.permit(
            user,
            address(this),
            permitInput.value,
            permitInput.deadline,
            permitInput.v,
            permitInput.r,
            permitInput.s
        );
    }
    // ...
}
```

Similarly, `swapDebt(...)` in `OdosDebtSwapAdapterV2.sol` unconditionally calls `delegationWithSig()`:

```
function swapDebt(
    DebtSwapParamsV2 memory debtSwapParams,
    CreditDelegationInput memory creditDelegationPermit,
    PermitInput memory collateralATokenPermit
) external nonReentrant whenNotPaused {
    // ...
    if (creditDelegationPermit.deadline != 0) {
        // @audit Unconditional delegationWithSig - vulnerable to front-running
        ICreditDelegationToken(creditDelegationPermit.debtToken).delegationWithSig(
            msg.sender,
            address(this),
            creditDelegationPermit.value,
            creditDelegationPermit.deadline,
            creditDelegationPermit.v,
            creditDelegationPermit.r,
            creditDelegationPermit.s
        );
    }
    // ...
}
```

The `executePendleSwap(...)` function calls `PendleSwapUtils.swapExactInput(...)`:

```
function executePendleSwap(
    address ptToken,
    uint256 ptAmount,
    address underlyingAsset,
    address pendleRouter,
    bytes memory swapData
) internal returns (uint256 actualUnderlyingOut) {
    // ...
    // @audit Exact-input swap - spends entire ptAmount unconditionally
    PendleSwapUtils.swapExactInput(ptToken, underlyingAsset, ptAmount, pendleRouter,
swapData);
    // ...
}
```

Affected code paths

1. `OdosWithdrawSwapAdapterV2.withdrawAndSwap(...)` → `_pullATokenAndWithdraw(...)`
2. `OdosLiquiditySwapAdapterV2.swapLiquidity(...)` → `_pullATokenAndWithdraw(...)`
3. `OdosRepayAdapterV2.repayWithCollateral(...)` → `_pullATokenAndWithdraw(...)`
4. `OdosDebtSwapAdapterV2.swapDebt(...)` → `delegationWithSig(...)`

Proof Of Concept

Dave wants to withdraw and swap his aUSDC using a permit signature for gas-efficient single-transaction flow.

1. Dave prepares a `permitInput` with signature `(v, r, s)` for his aUSDC:
   - `owner = Dave`
   - `spender = OdosWithdrawSwapAdapterV2`
   - `value = 1000e6`
   - `deadline = block.timestamp + 1 hour`
2. Dave submits transaction calling `withdrawAndSwap(...)` with this permit
3. Attack: Eve monitors the mempool and sees Dave's pending transaction
4. Eve extracts the permit parameters `(v, r, s, deadline, value)` from Dave's calldata
5. Eve front-runs by calling `aUSDC.permit(Dave, adapter, value, deadline, v, r, s)` directly
6. Eve's transaction executes first, consuming Dave's permit nonce
7. Dave's transaction reaches the block, calls `permit(...)` with the same parameters
8. Result: Dave's transaction reverts with "ERC2612: invalid signature" or similar error

Impact

All permit-enabled withdraw, repay, and debt-swap flows can be griefed. During market stress, blocking a repayment or collateral adjustment can directly lead to liquidations, causing significant financial loss to users.

Recommendation

Check existing allowance first; only call `permit()`/`delegationWithSig()` when still needed; wrap the call in `try/catch` and re-check allowance so that a front-run simply means the signature already took effect:

```
function _pullATokenAndWithdraw(
    address reserve,
    address user,
    uint256 amount,
    PermitInput memory permitInput
) internal returns (uint256) {
    (, , address aToken) = _getReserveData(reserve);

    if (permitInput.deadline != 0) {
        // Check if allowance already sufficient
        if (IERC20(aToken).allowance(user, address(this)) < amount) {
            try permitInput.aToken.permit(
                user, address(this), permitInput.value,
                permitInput.deadline, permitInput.v, permitInput.r, permitInput.s
            ) {} catch {
                // Permit may have been front-run - check if allowance is now
sufficient
                require(
                    IERC20(aToken).allowance(user, address(this)) >= amount,
                    "Permit failed and insufficient allowance"
                );
            }
        }
    }
    // ...
}
```

Status

Resolved

[H-05] OdosSwapUtils#executeSwapOperation & OdosWithdrawSwapAdapterV2#withdrawAndSwap -Same-token swaps steal adapter balances

Description

The `inputToken == outputToken` branch in `executeSwapOperation(...)` blindly sets `actualAmountReceived = exactOut` (the user-supplied minimum). Same-token swaps let attackers withdraw minimal collateral, request an arbitrarily large `minAmountToReceive`, and drain whatever balance the adapter holds for that token.

Vulnerability Details

The `executeSwapOperation(...)` function has a special case for same-token swaps that credits the user-provided `exactOut` value without verification:

```
function executeSwapOperation(
    IOdosRouterV2 router,
    address inputToken,
    address outputToken,
    uint256 maxIn,
    uint256 exactOut,
    bytes memory swapData
) internal returns (uint256 actualAmountSpent) {
    uint256 outputBalanceBefore = IERC20(outputToken).balanceOf(address(this));
    // ...
    uint256 outputBalanceAfter = IERC20(outputToken).balanceOf(address(this));
    uint256 actualAmountReceived;


    if (outputBalanceAfter >= outputBalanceBefore) {
        actualAmountReceived = outputBalanceAfter - outputBalanceBefore;
    } else if (inputToken == outputToken) {
        // @audit DANGEROUS: Credits user-controlled exactOut without verification
        // Same-asset flows intentionally net more tokens out than in
        // while returning minimal dust. Treat the caller-provided exactOut as the
credited amount.
        actualAmountReceived = exactOut;
```

```
    } else {
        revert InsufficientOutput(exactOut, 0);
    }
    // ...
}
```

After H-01 is fixed to return actualAmountReceived, an attacker can exploit withdrawAndSwap(...):

```
function withdrawAndSwap(
    WithdrawSwapParamsV2 memory withdrawSwapParams,
    PermitInput memory permitInput
) external nonReentrant whenNotPaused {
    // ...
    _pullATokenAndWithdraw(withdrawSwapParams.oldAsset, user,
withdrawSwapParams.oldAssetAmount, permitInput);


    // @audit amountReceived will be the attacker-controlled minAmountToReceive for
same-token swaps
    uint256 amountReceived = _executeAdaptiveSwap(
        IERC20Detailed(withdrawSwapParams.oldAsset),
        IERC20Detailed(withdrawSwapParams.newAsset),  // @audit same as oldAsset
        withdrawSwapParams.oldAssetAmount,
        withdrawSwapParams.minAmountToReceive,  // @audit attacker sets this to the
adapter's balance
        withdrawSwapParams.swapData
    );
    // ...
    // @audit Transfers attacker-controlled amount to attacker
    IERC20(withdrawSwapParams.newAsset).safeTransfer(user, amountReceived);
}
```

Proof Of Concept

Adapter contract holds 1,000 USDC dust from previous operations. Attacker Frank exploits this.

1. Frank notices OdosWithdrawSwapAdapterV2 holds 1,000 USDC dust

2. Frank has a tiny position: 1 aUSDC (worth $0.000001)
3. Frank calls `withdrawAndSwap(...)` with:
   - `oldAsset = USDC`
   - `newAsset = USDC` (same token!)
   - `oldAssetAmount = 1` (1 wei of aUSDC)
   - `minAmountToReceive = 1000e6` (1,000 USDC - the adapter's balance)
4. The adapter pulls 1 aUSDC from Frank and withdraws 1 USDC from Aave
5. `_executeAdaptiveSwap(...)` calls `OdosSwapUtils.executeSwapOperation(...)`:
   - `inputToken = outputToken = USDC`
   - The same-token branch triggers: `actualAmountReceived = exactOut = 1000e6`
6. After H-01 fix, `amountReceived = 1000e6` is returned
7. The adapter executes `safeTransfer(Frank, 1000e6)`
8. Result: Frank receives 1,000 USDC while only providing 1 wei of collateral

Impact

Every adapter that routes through `OdosSwapUtils` becomes vulnerable to instant theft of whatever ERC20 balances it holds. Attackers can repeatedly sweep dust, front-run owner `rescueToken()` calls, or drain any tokens accidentally transferred to the adapters.

Recommendation

Disallow same-token swaps in production code:

```
function executeSwapOperation(
    IOdosRouterV2 router,
    address inputToken,
    address outputToken,
    uint256 maxIn,
    uint256 exactOut,
    bytes memory swapData
) internal returns (uint256 actualAmountSpent) {
    // Prevent same-token swap exploits
    if (inputToken == outputToken) {
        revert("Same token swaps not supported");
    }
    }
```

Status

Resolved

Medium

[M-01] DStakeTokenV2.sol#migrateCore() - Router Migration During Active Shortfall Causes Instant Share Price Inflation

Description

The `DStakeTokenV2` contract allows migration to a new router while an active shortfall exists, causing the new router to start with zero shortfall and instantly inflating the share price.

Vulnerability Details

The `DStakeTokenV2.totalAssets()` function calculates net assets by subtracting the router's current shortfall:

```
function totalAssets() public view virtual override returns (uint256) {
    uint256 grossAssets = _grossTotalAssets();
    if (grossAssets == 0) {
        return 0;
    }

    uint256 shortfall = address(router) == address(0) ? 0 : router.currentShortfall();
    return shortfall >= grossAssets ? 0 : grossAssets - shortfall;
}
```

However, the `migrateCore()` function does not verify that the shortfall has been cleared before migration:

```
function migrateCore(address newRouter, address newCollateralVault) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (newRouter == address(0) || newCollateralVault == address(0)) {
        revert ZeroAddress();
```

```
    }



    IDStakeRouterV2 routerCandidate = IDStakeRouterV2(newRouter);

                                    if        (routerCandidate.collateralVault()        !=
IDStakeCollateralVaultV2(newCollateralVault)) {

                        revert   RouterCollateralMismatch(newRouter,   newCollateralVault,
address(routerCandidate.collateralVault()));

    }



    if (routerCandidate.dStakeToken() != address(this)) {

                                revert    RouterTokenMismatch(newRouter,    address(this),
routerCandidate.dStakeToken());

    }



    router = routerCandidate;

    collateralVault = IDStakeCollateralVaultV2(newCollateralVault);

    // No check for router.currentShortfall() == 0

}
```

When governance migrates to a new router while the old router has an active shortfall, the new router starts with `settlementShortfall == 0` by default. This causes `totalAssets()` to immediately jump from `(grossAssets - oldShortfall)` to `grossAssets`, artificially inflating the share `price`. Consider a scenario where the protocol has $10M in gross assets with a $2M shortfall recorded. The share price reflects $8M in net value. After migration to a new router with zero shortfall, `totalAssets()` suddenly returns $10M, causing a 25% instant share price increase. This allows depositors who enter immediately after migration to capture value that should have

been socialized across existing shareholders until the shortfall was legitimately resolved. The issue breaks fair-value accounting principles and enables value extraction through privileged knowledge of pending migrations.

Impact

The vulnerability enables instant share price manipulation through administrative action, creating an unfair advantage for depositors with advance knowledge of router migrations. Existing shareholders who held positions during the shortfall period effectively subsidize new depositors who enter after migration, violating the principle that losses should be socialized across all participants. The impact extends beyond individual unfairness to protocol reputation, as the sudden share price jump appears as accounting manipulation rather than legitimate value recovery. This undermines trust in the protocol's financial transparency and could trigger regulatory scrutiny. The risk is particularly acute because router migrations are governance-controlled events that might be predictable to insiders, creating opportunities for front-running or coordinated extraction of value from existing shareholders.

Recommendation

Add a shortfall verification requirement in the `migrateCore()` function to prevent migration while losses remain unresolved. The check should revert if any shortfall exists, forcing governance to either clear the shortfall through capital injection or explicitly account for it via the new router's `recordShortfall()` function before migration completes. Implement the fix by adding a requirement at the beginning of the migration function that verifies the current router's shortfall is zero. This simple check ensures that any recorded losses are properly resolved or carried forward before the accounting context changes, maintaining fair-value integrity across router upgrades. Additionally, consider emitting an event when shortfalls are detected during migration attempts to provide transparency into why migrations might be blocked and encourage proper shortfall resolution procedures.

```
function migrateCore(address newRouter, address newCollateralVault) external
onlyRole(DEFAULT_ADMIN_ROLE) {

    if (newRouter == address(0) || newCollateralVault == address(0)) {
```

```
        revert ZeroAddress();
    }



    // Require shortfall to be cleared before migration
    if (address(router) != address(0)) {
        require(router.currentShortfall() == 0, "ShortfallActive");
    }



    IDStakeRouterV2 routerCandidate = IDStakeRouterV2(newRouter);
    // ... rest of migration logic
}
```

Status

Resolved

Low

[L-01] IssuerV2_1 & RedeemerV2#revokeRole/renounceRole - Admin role can be removed permanently

Description

Both contracts inherit OpenZeppelin AccessControl and expose revokeRole()/renounceRole() without safeguards. The last holder of DEFAULT_ADMIN_ROLE can be removed, leaving the contracts ownerless and preventing future governance actions.

Vulnerability Details

Both contracts inherit OpenZeppelin AccessControl and expose revokeRole()/renounceRole() without safeguards. The last holder of DEFAULT_ADMIN_ROLE can be removed, leaving the contracts ownerless and preventing future governance actions.

```
contract IssuerV2_1 is AccessControl, OracleAware, ReentrancyGuard, Pausable {
    // ...
    constructor(
        address _collateralVault,
        address _dstable,
        IPriceOracleGetter oracle
    ) OracleAware(oracle, oracle.BASE_CURRENCY_UNIT()) {
        // ...
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        // @audit No protection against revoking/renouncing the last admin
    }
}
```

OpenZeppelin's AccessControl allows:

- Any admin to call revokeRole(DEFAULT_ADMIN_ROLE, lastAdmin)

- Any admin to call renounceRole(DEFAULT_ADMIN_ROLE, theirOwnAddress)

Impact

Losing the final admin freezes every privileged function. The protocol would be unable to respond to incidents, adjust parameters, or upgrade dependencies, potentially leading to catastrophic fund loss.

Recommendation

Override `revokeRole()` and `renounceRole()` to require that at least one admin remains:

Status

Acknowledged

[L-02] IssuerV2_1 constructors & DLend adapter constructors - Missing zero-address validation for dependencies

Description

IssuerV2_1 and multiple DLend adapters assign immutable dependencies in their constructors without checking for zero addresses, potentially bricking contracts on deployment.

Vulnerability Details

IssuerV2_1 constructor does not validate input parameters (unlike RedeemerV2, which does):

```
// IssuerV2_1.sol
constructor(
    address _collateralVault,
    address _dstable,
    IPriceOracleGetter oracle
) OracleAware(oracle, oracle.BASE_CURRENCY_UNIT()) {
    // @audit No zero-address validation (compare to RedeemerV2 which has it)
    collateralVault = CollateralVault(_collateralVault);
    dstable = IMintableERC20(_dstable);
    dstableDecimals = dstable.decimals();  // @audit Will revert if _dstable is
address(0)
    // ...
}
```

Similarly, DLend adapter constructors lack validation in BaseOdosSwapAdapter

```
constructor(IPoolAddressesProvider addressesProvider, address pool) Ownable(msg.sender)
{
    // @audit No validation
    ADDRESSES_PROVIDER = addressesProvider;
    POOL = IPool(pool);
}
```

Impact

A misconfigured deployment wastes gas and yields an unusable contract. Because these references are immutable, redeployment is required, delaying upgrades or launches.

Recommendation

Add validation consistent across both contracts.

Status

Resolved

[L-03] AmoManagerV2#repayWithPermit- Admin-only permit can still be front-run

Description

repayWithPermit(...) calls IERC20Permit.permit() unconditionally. Although restricted to AMO_DECREASE_ROLE, a mempool observer can consume the signature, causing the admin transaction to revert and requiring a retry.

Vulnerability Details

The repayWithPermit(...) function executes permit unconditionally:

```
function repayWithPermit(
    address wallet,
    address asset,
    uint256 amount,
    uint256 maxDebtBurned,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external onlyRole(AMO_DECREASE_ROLE) nonReentrant {
    // @audit Unconditional permit call
    IERC20Permit(asset).permit(wallet, address(this), amount, deadline, v, r, s);
    repayFrom(wallet, asset, amount, maxDebtBurned);
}
```

Impact

Operational inconvenience for AMO operators. No direct fund loss, but wasted gas and potential missed opportunities during time-sensitive operations.

Recommendation

Wrap the permit call in try/catch and verify allowance after failures:

```
function repayWithPermit(
    // ...
```

```
) external onlyRole(AMO_DECREASE_ROLE) nonReentrant {
    try IERC20Permit(asset).permit(wallet, address(this), amount, deadline, v, r, s) {
        // Permit successful
    } catch {
        // Permit may have been front-run - check if allowance is sufficient
        require(
            IERC20(asset).allowance(wallet, address(this)) >= amount,
            "Permit failed and insufficient allowance"
        );
    }
    repayFrom(wallet, asset, amount, maxDebtBurned);
}
```

Status

Resolved

[L-04] BaseOdosSwapAdapter#_conditionalRenewAllowance & OdosDebtSwapAdapterV2#constructor - Infinite approvals incompatible with UNI/COMP-style tokens

Description

The adapters approve `type(uint256).max` when refreshing allowances and during constructor pre-approvals. Tokens such as UNI and COMP reject approvals above `uint96`, causing allowance renewals or even adapter deployment to revert once those assets are listed.

Vulnerability Details

The `_conditionalRenewAllowance(...)` function approves max uint256:

```
function _conditionalRenewAllowance(address asset, uint256 minAmount) internal {
    uint256 allowance = IERC20(asset).allowance(address(this), address(POOL));
    if (allowance < minAmount) {
        // @audit type(uint256).max will revert for UNI/COMP tokens
        IERC20(asset).safeApprove(address(POOL), type(uint256).max);
    }}
```

Similarly, `OdosDebtSwapAdapterV2` constructor pre-approves all reserves:

```
constructor(
    // ...
) BaseOdosBuyAdapterV2(addressesProvider, pool, _swapRouter, _pendleRouter) {
    // ...
    address[] memory reserves = POOL.getReservesList();
    for (uint256 i = 0; i < reserves.length; i++) {
        // @audit Will revert if any reserve is UNI/COMP-style token
        IERC20WithPermit(reserves[i]).safeApprove(address(POOL), type(uint256).max);
    }
}
```

UNI token explicitly restricts approvals:

```
function approve(address spender, uint rawAmount) external returns (bool) {
    uint96 amount;
    if (rawAmount == uint(-1)) {
        amount = uint96(-1);
    } else {
        // @audit Reverts if rawAmount > uint96.max
        require(rawAmount <= uint96(-1), "Uni::approve: amount exceeds 96 bits");
        amount = uint96(rawAmount);
    }
    // ...
}
```

Impact

Adapters cannot list or interact with several blue-chip governance tokens, and the debt swap adapter could fail deployment entirely if a restricted token exists in the reserve list.

Recommendation

Use `type(uint96).max` instead of `type(uint256).max`:

```
function _conditionalRenewAllowance(address asset, uint256 minAmount) internal {
    uint256 allowance = IERC20(asset).allowance(address(this), address(POOL));
    if (allowance < minAmount) {
        // Use uint96 max for compatibility with governance tokens
        IERC20(asset).safeApprove(address(POOL), type(uint96).max);
    }
}
```

Status

Resolved

[L-05] DStakeIdleVault.sol$setEmissionSchedule() - Emission Schedule Does Not Validate Sufficient Reserve Funding

Description

The `DStakeIdleVault` allows setting emission schedules without verifying that sufficient reward reserves exist to cover the entire emission period.

Vulnerability Details

The `setEmissionSchedule()` function accepts parameters for emission start time, end time, and per-second emission rate, but does not validate that the current `rewardReserve` contains enough tokens to sustain emissions for the entire configured duration:

```
function setEmissionSchedule(uint64 start, uint64 end, uint256 rate) external onlyRole(REWARD_MANAGER_ROLE) {
    if (end != 0 && end <= start) {
        revert InvalidEmissionWindow();
    }
    _accrue();

    emissionStart = start;
    emissionEnd = end;
    emissionPerSecond = rate;
    lastEmissionUpdate = uint64(block.timestamp);

    emit EmissionScheduleSet(start, end, rate);
}
```

This creates a situation where emissions can be configured for 30 days but only 11 days worth of rewards are funded. When the reserve depletes at day 11, the `_pendingEmission()` function caps the return value at the remaining reserve, causing

emissions to silently stop 19 days early. The emission calculation includes a safety cap but does not alert operators or users that the reserve is insufficient. Consider a configuration where emissions are set to distribute 1 token per second for 30 days, requiring 2,592,000 tokens total, but `only` 1,000,000 tokens are funded. After approximately 11.57 days, the reserve hits zero, and `_pendingEmission()` begins returning zero for all subsequent calls. Users continue depositing into the vault expecting 30 days of rewards, but receive nothing after day 11, resulting in significantly lower APY than advertised. The situation can also occur if the reward manager withdraws unreleased rewards mid-stream using withdrawUnreleasedRewards(), deliberately or accidentally creating a shortfall that causes emissions to stop before the configured end time.

Impact

The lack of upfront validation allows emission schedules to be configured with insufficient funding, leading to premature termination of rewards and user disappointment. This creates an unfair situation where early depositors receive rewards while later depositors receive nothing, despite both groups having reasonable expectations based on the configured schedule.

Recommendation

Implement validation logic in `setEmissionSchedule()` to verify that the current reward reserve contains sufficient tokens to cover the entire emission period at the specified rate. Calculate the total required reserves as `(end - start) * rate` and compare against the current `rewardReserve` balance, reverting if insufficient funds exist.

Status

Resolved

QA

[Q-01] IssuerV2_1#setCollateralVault -Missing zero-address validation for admin updates

Description

`setCollateralVault(...)` updates the vault address without validating it, so an accidental `address(0)` bricks all minting operations until fixed.

Vulnerability Details

Missing zero address checks in `setCollateralVault`

```
function setCollateralVault(address _collateralVault) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    // @audit No validation - address(0) accepted
    collateralVault = CollateralVault(_collateralVault);
    emit CollateralVaultSet(_collateralVault);
}
```

Impact

An accidental zero-address assignment temporarily disables all minting operations, causing protocol downtime and potential revenue loss during high-demand periods. While reversible by admin action, the lack of validation creates unnecessary operational risk.

Recommendation

Add zero-address check:

```
function setCollateralVault(address _collateralVault) external
onlyRole(DEFAULT_ADMIN_ROLE) {
    if (_collateralVault == address(0)) {
        revert CannotBeZeroAddress();
    }
```

```
    collateralVault = CollateralVault(_collateralVault);
    emit CollateralVaultSet(_collateralVault);
}
```

Status

Resolved

[Q-02] RedeemerV2#setFeeReceiver -Redundant state writes waste gas

Description

The admin can call setFeeReceiver(...) with the existing address, incurring an unnecessary SSTORE and emitting a misleading event.

Vulnerability Details

```
function setFeeReceiver(address _newFeeReceiver) external onlyRole(DEFAULT_ADMIN_ROLE)
{
    if (_newFeeReceiver == address(0)) {
        revert CannotBeZeroAddress();
    }
    // @audit No check if _newFeeReceiver == feeReceiver (redundant write)
    address oldFeeReceiver = feeReceiver;
    feeReceiver = _newFeeReceiver;
    emit FeeReceiverUpdated(oldFeeReceiver, _newFeeReceiver);
}
```

Impact

Redundant state writes waste gas and pollutes event logs with no-op changes, making it harder to track actual fee receiver updates during audits or investigations.

Recommendation

Add early return when value unchanged:

```
function setFeeReceiver(address _newFeeReceiver) external onlyRole(DEFAULT_ADMIN_ROLE)
{
    if (_newFeeReceiver == address(0)) {
        revert CannotBeZeroAddress();
    }
    if (_newFeeReceiver == feeReceiver) {
        return; // No change needed
    }

}
```

Status

Acknowledged

[Q-03] IssuerV2_1 & RedeemerV2 inheritance - Redundant AccessControl declaration

Description

Both contracts inherit AccessControl directly and via OracleAware, adding noise and potential confusion, even though Solidity linearization resolves it.

Vulnerability Details

```
// IssuerV2_1.sol
contract IssuerV2_1 is AccessControl, OracleAware, ReentrancyGuard, Pausable {
    // @audit AccessControl inherited both directly and via OracleAware
}
// OracleAware.sol
abstract contract OracleAware is AccessControl {
    // @audit Already inherits AccessControl
}
```

Impact

Redundant inheritance declarations create unnecessary confusion for developers reading the code, potentially leading to misunderstandings about contract structure and wasting onboarding time.

Recommendation

Remove direct AccessControl inheritance

```
// IssuerV2_1.sol
/// @dev Inherits AccessControl through OracleAware
contract IssuerV2_1 is OracleAware, ReentrancyGuard, Pausable {
    // ...
}
```

Status

Acknowledged

[Q-04] AmoManagerV2#setTolerance - No bounds checking

Description

setTolerance(...) accepts any value, so admins can accidentally set zero (bricking AMO flows) or an extremely high tolerance (weakening invariant checks).

Vulnerability Details

```
function setTolerance(uint256 newTolerance) external onlyRole(DEFAULT_ADMIN_ROLE) {
    // @audit No bounds validation
    uint256 oldTolerance = tolerance;
    tolerance = newTolerance;
    emit ToleranceSet(oldTolerance, newTolerance);
}
```

The tolerance is used in invariant checks:

```
// In increaseAmoSupply, decreaseAmoSupply, borrowTo, repayFrom
if (
    actualDebtIncrease + tolerance < expectedDebtFromDstable ||
    actualDebtIncrease > expectedDebtFromDstable + tolerance
) {
    revert InvariantViolation(expectedDebtFromDstable, actualDebtIncrease);
}
```

Impact

Unbounded tolerance values can either brick all AMO operations (if set to zero) or allow significant value discrepancies to go undetected (if set too high), potentially leading to protocol downtime or undetected value leakage.

Recommendation

Enforce reasonable bounds:

Status

Acknowledged

[Q-05] All contracts#pragma - Floating compiler version introduces non-determinism

Description

Virtually every contract uses `pragma solidity ^0.8.20;`, allowing compilation with any 0.8.x release. Different compiler versions can change bytecode, gas costs, or behavior.

Vulnerability Details

All contracts use a floating pragma:

```
// OdosSwapUtils.sol, BaseOdosSwapAdapter.sol, IssuerV2_1.sol, etc.
pragma solidity ^0.8.20;  // @audit Allows 0.8.20, 0.8.21, 0.8.22, etc.
```

Impact

Floating compiler versions introduce non-determinism in bytecode generation, potentially creating discrepancies between audited and deployed code. This weakens audit assurance and may result in security gaps if compiler bugs or behavioural changes exist between versions.

Recommendation

Lock to specific version:

```
pragma solidity 0.8.20;  // Exact version, not floating
```

Status

Acknowledged

[Q-06] SwapExecutorV2#executeSwapExactInput/executeSwapExactOutput -Dead REGULAR_SWAP branch adds noise

Description

By construction, `_executeComposedSwap*()` only calls `SwapExecutorV2` for PT swaps, yet the executor recomputes the swap type and includes an unreachable REGULAR branch.

Vulnerability Details

```
// SwapExecutorV2.sol
function executeSwapExactInput(ExactInputParams memory params) internal returns
(uint256 actualOutputAmount) {
    ISwapTypes.SwapType swapType = PendleSwapLogic.determineSwapType(params.inputToken,
params.outputToken);



    // @audit This branch is unreachable - callers already filter for PT swaps
    if (swapType == ISwapTypes.SwapType.REGULAR_SWAP) {
        return OdosSwapUtils.executeSwapOperation(...);
    }
    // ...
}
```

The caller `BaseOdosSellAdapterV2._executeAdaptiveSwap(...)` already handles regular swaps directly:

```
function _executeAdaptiveSwap(...) internal returns (uint256 amountReceived) {
    ISwapTypes.SwapType swapType = PendleSwapLogic.determineSwapType(tokenIn,
tokenOut);



    if (swapType == ISwapTypes.SwapType.REGULAR_SWAP) {
        // @audit Regular swaps handled here, never reach SwapExecutor
        return _executeDirectOdosExactInput(...);
    }
```

```
    // Only PT swaps reach _executeComposedSwapExactInput which calls SwapExecutor
    amountReceived = _executeComposedSwapExactInput(...);
    // ...
}
```

Impact

Dead code branches create confusion about the intended execution paths and may mislead developers into routing transactions through incorrect code paths, potentially bypassing important validation logic.

Recommendation

Remove unreachable branch

Status

Resolved

[Q-07] IOdosWithdrawSwapAdapterV2#WithdrawSwapParamsV2 - allBalanceOffset stored as uint256 instead of bool

Description

The allBalanceOffset field is treated strictly as a boolean flag but is defined as uint256, bloating calldata and masking intent.

Vulnerability Details

```
// IOdosWithdrawSwapAdapterV2.sol
struct WithdrawSwapParamsV2 {
    // ...
    uint256 allBalanceOffset;  // @audit Used as bool, stored as uint256
    // ...
}
```

Usage in withdrawAndSwap(...):

```
function withdrawAndSwap(
    WithdrawSwapParamsV2 memory withdrawSwapParams,
    PermitInput memory permitInput
) external nonReentrant whenNotPaused {
    // ...
    // @audit Only checks != 0, never uses numeric value
    if (withdrawSwapParams.allBalanceOffset != 0) {
        uint256 balance = IERC20(aToken).balanceOf(user);
        withdrawSwapParams.oldAssetAmount = balance;
    }
    // ...
}
```

Impact

Using uint256 for a boolean flag wastes calldata gas and obscures the intended usage, leading to developer confusion during integration and unnecessary gas costs on every transaction.

Recommendation

Change to bool

```
struct WithdrawSwapParamsV2 {
    // ...
    bool useAllBalance;  // Clear intent
    // ...
```

Status

Acknowledged

[Q-08] GenericERC4626ConversionAdapter.sol - Missing Emergency Withdrawal in GenericERC4626ConversionAdapter

Description

The `GenericERC4626ConversionAdapter` lacks an emergency withdrawal function to recover stuck dStable tokens that may remain after failed or partial deposits.

Impact

The missing emergency withdrawal `capability` means that any dStable tokens that become stuck in the adapter due to unexpected vault behavior are permanently lost to the protocol. The lack of a recovery mechanism creates an asymmetric risk where downside scenarios trap funds, but upside scenarios provide no corresponding benefit.

Recommendation

Implement an `emergencyWithdraw()` function in `GenericERC4626ConversionAdapter` matching the pattern used in `MetaMorphoConversionAdapter`. The function should accept a token address and amount, verify the caller has `DEFAULT_ADMIN_ROLE`, and transfer the specified tokens to the caller.

Status

Acknowledged

[Q-09] GenericERC4626ConversionAdapter.sol, MetaMorphoConversionAdapter.sol, WrappedDLendConversionAdapter.sol - Adapter Functions Callable by Arbitrary Users

Description

Adapter contracts expose deposit and withdrawal functions without restricting callers to the router, allowing users to interact directly with adapters in unintended ways.

Impact

Users who mistakenly interact with adapters directly experience permanent loss of their dStable tokens, receiving strategy shares that are sent to the collateral vault and become irretrievable. This creates a user experience hazard where confusion about the proper interaction flow leads to financial loss.

Recommendation

Restrict adapter deposit and withdrawal functions to only be callable by the router address using a simple access control check.

Status

Resolved

[Q-10] DStakeCollateralVaultV2.sol#rescueToken() - Collateral Vault Blocks Rescue of dStable Despite Architecture Not Using It

Description

`DStakeCollateralVaultV2` prevents rescuing dStable tokens despite the architecture never intentionally holding dStable in the collateral vault.

Impact

The overly restrictive rescue function causes permanent loss of any dStable tokens accidentally sent to the collateral vault, with no recovery mechanism available to the protocol. Users who mistakenly send dStable directly to the collateral vault address experience irreversible loss, unable to retrieve their funds even with governance intervention.

Recommendation

Remove the dStable restriction from the `rescueToken()` function in `DStakeCollateralVaultV2`, allowing governance to rescue dStable that was never meant to be there.

Status

Resolved

[Q-11] GenericERC4626ConversionAdapter.sol#depositIntoStrategy() - Missing Allowance Reset in GenericERC4626ConversionAdapter

Description

`GenericERC4626ConversionAdapter` does not reset token allowance to zero after deposit operations, unlike other adapters in the system

Impact

The impact of this issue is minimal since the adapter architecture does not hold balances between operations and approvals are set to exact amounts before each deposit. However, the lack of approval reset represents a deviation from security best practices and creates unnecessary trust assumptions about vault behavior.

Recommendation

Add an approval reset to zero after the deposit operation completes in `GenericERC4626ConversionAdapter`, matching the pattern used in other adapters. Insert `IERC20(dStable).forceApprove(address(vault), 0);` after the deposit call and before the function returns.

Status

Resolved

[Q-12] RewardClaimable.sol#compoundRewards() - Reward Compounding Requires Threshold Payment to Recover Omitted Tokens

Description

The `RewardClaimable` compounding mechanism requires re-paying the exchange threshold to recover rewards from tokens accidentally omitted from the rewardTokens array.

Impact

Operators who make mistakes in reward token array construction face a disproportionate cost to recover from the error, potentially discouraging proper reward distribution. The issue affects operational efficiency rather than security but creates unnecessary friction in the reward distribution process.

Recommendation

Implement a separate recovery function that allows distributing already-claimed rewards without requiring a fresh exchange asset deposit.

Status

Acknowledged

[Q-13] DStakeRouterV2.sol, DStakeTokenV2.sol, DStakeCollateralVaultV2.sol - Missing Last Admin Protection in AccessControl Inheritance

Description

Several contracts inherit OpenZeppelin's AccessControl but do not override `revokeRole()` and `renounceRole()` to prevent removal of the last admin.

Impact

Loss of the last admin role results in permanent administrative paralysis across affected contracts. The protocol would be unable to respond to security incidents, adjust parameters in response to market conditions, onboard new yield strategies, or perform routine maintenance operations.

Recommendation

Override the `revokeRole()` and `renounceRole()` functions in all contracts using AccessControl to include checks that prevent removal of the last admin.

Status

Acknowledged

Centralisation

The dTRINITY project values community-centered and transparency over centralisation.

dTRINITY emphasizes a community-centric approach while maintaining transparency. Although some critical functions remain centralized for enhanced security and functionality, the project strives to balance decentralization with internal team accountability. This blend ensures operational security while keeping the protocol aligned with decentralized values.

Conclusion

After Hashlock's analysis, the dTRINITY project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au