hexens x USUAL

Security Review Report for Usual

November 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This audit covers USD0x, a synthetic USD-denominated yield-accruing asset collateralized by a mix of Real World Asset (RWA) tokens and USDC. The protocol employs a flexible collateralization model in which USD0x's price is derived from both the value of the underlying collateral and the asset's total supply.

Our review was conducted over the course of one week and involved a comprehensive analysis of the Solidity smart contracts. Core components include a modular registry system, oracle aggregation, role-based access control, and an orderbook-driven "swapper engine" used to facilitate USD0x/USDC liquidity.

During the assessment, we identified one medium-severity vulnerability involving insufficient user-side slippage protection when using the swapper engine with partial fills. In addition, we reported one low-severity issue and three informational findings.

All identified issues were either remediated or acknowledged by the development team and subsequently verified by our auditors.

Following the remediation phase, we conclude that the protocol's overall security posture and code quality have been notably strengthened as a result of this audit.

3. Security Review Details

▪ Review Led by

Trung Dinh, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/usual-dao/USD0x-protocol

📌 Commit: `c1cc5c7b98ba7aff876467cb112cc4af5e8d1918`

The issues described in this report were fixed in the following commit:

🔗 https://github.com/usual-dao/usd0x-protocol/pull/29

📌 Commit: `2b8352713318b14e34f8f0e4ab0aaa0172139d58`

▪ Changelog

3 November2025	Audit start
17 November 2025	Initial report
18 November 2025	Revision received
20 November 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

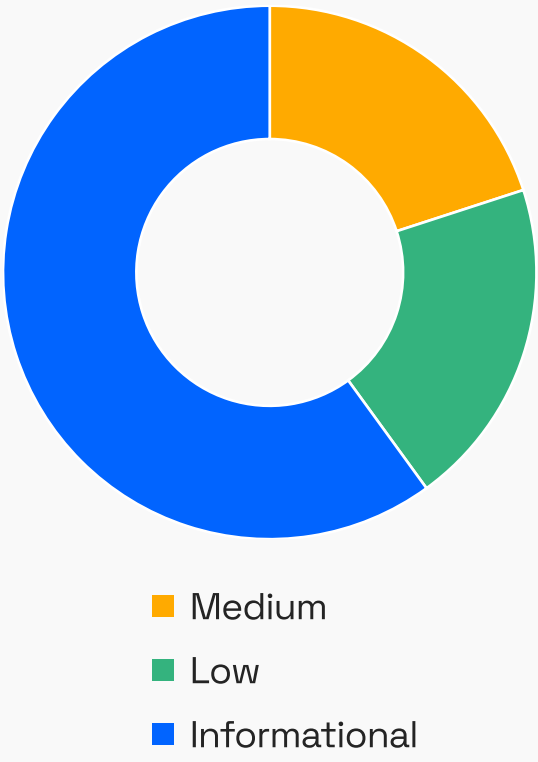
- **Issue Symbolic Codes**

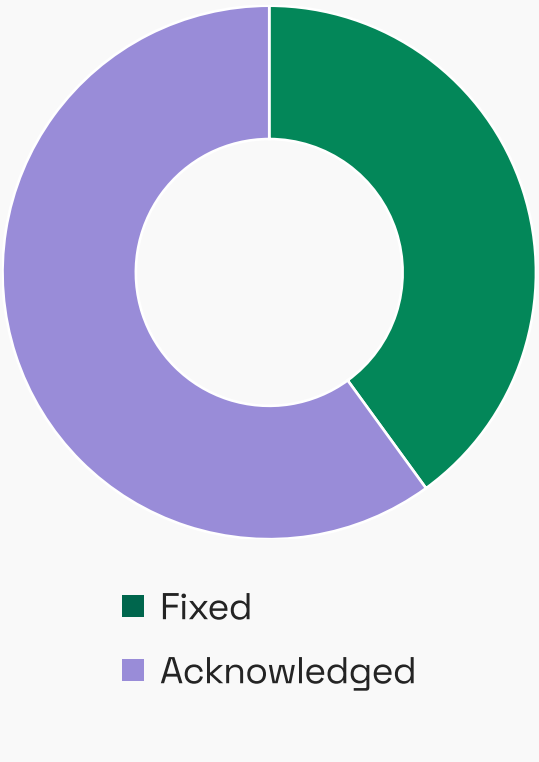
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	0
Medium	1
Low	1
Informational	3
Total:	**5**





6. Weaknesses

This section contains the list of discovered weaknesses.

USL3-1 | Insufficient Slippage Check in provideUSDCReceiveUSD0X Function

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

src/swapperEngine/SwapperEngine.sol#L436-L450

Description:

The **SwapperEngine.provideUSDCReceiveUSD0X()** function enables users to provide USDC tokens in exchange for USD0X tokens by matching with existing orders.

This function includes an input parameter, **maxUsdcAmountToSpend**, which serves as a slippage control to ensure users do not spend more USDC than intended. It also accepts a boolean flag, **partialMatchingAllowed**, indicating whether partial order matching is permitted.

The issue arises when the **partialMatchingAllowed** flag is set to **true**, as this affects the intended semantics of the **maxUsdcAmountToSpend** parameter. The **maxUsdcAmountToSpend** value is designed to represent the maximum USDC amount the user is willing to spend to receive the full target amount specified by **amountUSD0XToReceive**. However, when partial matching is allowed, the proportion between the received and spent amounts changes, and thus the same **maxUsdcAmountToSpend** threshold should not apply.

Example:

1. Alice wants to receive 1,000,000 USD0X by spending at most 1,000,000 USDC.
   She calls **provideUSDCReceiveUSD0X()** with:
   - **amountUSD0XToReceive = 1e6 * 1e18**
   - **maxUsdcAmountToSpend = 1e6 * 1e6**
   - **partialMatchingAllowed = true**

Her goal is to allow partial matching so she can still purchase smaller amounts of USD0X at a 1:1 rate (e.g., 1,000 USDC for 1,000 USD0X).

2. During execution, suppose the exchange rate shifts to 1 USDC = 0.99 USD0X.
   The function may spend the full 1,000,000 USDC but only deliver 990,000 USD0X, causing Alice to incur a 10,000 USD0X loss compared to her expected rate at the time of order creation.

```
    // Check if total USDC spent exceeds the maximum allowed
    if (totalUsdcSpent > maxUsdcAmountToSpend) {
        revert UsdcSpentExceedsMaximum();
    }


    // Transfer USD0X from this contract to the recipient
    $.usd0xToken.safeTransfer(recipient, totalUSD0XTaken);


    // Revert if partial matching is not allowed and we haven't taken all of the USD0X
    if (
        !partialMatchingAllowed && totalUSD0XTaken != amountUSD0XToReceive
            || totalUSD0XTaken == 0
    ) {
        revert AmountTooLow();
    }
```

Remediation:

To ensure the slippage check correctly reflects partial matches, the logic can be adjusted as follows:

```
    if (
        !partialMatchingAllowed && totalUSD0XTaken != amountUSD0XToReceive ||
        totalUSD0XTaken == 0
    ) {
        revert AmountTooLow();
    }

++  if (partialMatchingAllowed) {
++      uint modifiedMaxUsdcAmountToSpend = Math.mulDiv(
++          maxUsdcAmountToSpend,
++          totalUSD0XTaken,
++          amountUSD0XToReceive,
++          Math.Rounding.Floor
++      );

++      if (totalUsdcSpent > modifiedMaxUsdcAmountToSpend) {
++          revert UsdcSpentExceedsMaximum();
++      }
++  }
```

This modification dynamically scales the **maxUsdcAmountToSpend** based on the proportion of **totalUSD0XTaken** relative to the intended **amountUSD0XToReceive**, ensuring a consistent slippage threshold even when partial matching is enabled.

USL3-5 | The function getUSD0XPriceInUSD does not fully apply rounding up when roundingUp is set to true

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/token/USD0X.sol#L313-L317

Description:

The function **USD0X.getUSD0XPriceInUSD()** calculates the USD price of one USD0X token based on the total USD value of all collateral backing it. It accepts a boolean parameter **roundingUp** to indicate whether the result should be rounded up.

```
function getUSD0XPriceInUSD(bool roundingUp) external view returns (uint256) {
    USD0XTokenStorageV0 storage $ = _usd0XTokenStorageV0();
    address[] memory collateralTokens = $.tokenMapping.getAllUsd0XCollateralTokens();

    uint256 wadCollateralBackingInUSD = 0;
    for (uint256 i = 0; i < collateralTokens.length;) {
        address collateralToken = collateralTokens[i];
        uint256 collateralTokenPriceInUSD = uint256($.oracle.getPrice(collateralToken));
        uint8 decimals = IERC20Metadata(collateralToken).decimals();
        wadCollateralBackingInUSD += Math.mulDiv(
            collateralTokenPriceInUSD,
            IERC20(collateralToken).balanceOf($.collateralTreasury),
            10 ** decimals
        );
        unchecked {
            ++i;
        }
    }

    // Handle edge cases
    uint256 currentTotalSupply = totalSupply();
    // slither-disable-next-line incorrect-equality
    if (currentTotalSupply == 0 || wadCollateralBackingInUSD == 0) {
        return 0;
    }
```

```
    if (roundingUp) {
        return Math.mulDiv(
            wadCollateralBackingInUSD, SCALAR_ONE, currentTotalSupply, Math.Rounding.Ceil
        );
    } else {
        return Math.mulDiv(
            wadCollateralBackingInUSD, SCALAR_ONE, currentTotalSupply, Math.Rounding.Floor
        );
    }
}
```

However, within the collateral iteration, the variable **wadCollateralBackingInUSD** is accumulated using **Math.mulDiv()** without specifying a rounding mode. By default, this performs rounding down. As a result, even when **roundingUp** is **true**, the intermediate calculations round down, causing the final price to not fully reflect an upward-rounded result.

Remediation:

To ensure consistent rounding behavior, apply the **Math.Rounding.Ceil** option to the **Math.mulDiv()** operation inside the collateral accumulation loop when **roundingUp** is true. For example:

```
wadCollateralBackingInUSD += Math.mulDiv(
    collateralTokenPriceInUSD,
    IERC20(collateralToken).balanceOf($.collateralTreasury),
    10 ** decimals,
    roundingUp ? Math.Rounding.Ceil : Math.Rounding.Floor
);
```

USL3-2 | Redundant Blacklist Check in _depositUSD0X Function

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/swapperEngine/SwapperEngine.sol#L202-L204

Description:

In the **SwapperEngine._depositUSD0X()** function, lines 202–204 perform a blacklist check that reverts if **msg.sender** is blacklisted by the **USD0X** token:

```
if ($.usd0xToken.isBlacklisted(msg.sender)) {
    revert NotAuthorized();
}
```

However, this check is redundant. Later in the same function (line 215), the contract calls **usd0xToken.safeTransferFrom(msg.sender, address(this), amountToDeposit)**.

The **USD0X** token's **transferFrom()** function internally invokes the **_update()** function, which already enforces a blacklist validation on both the sender and receiver addresses:

```
function _update(address from, address to, uint256 amount)
    internal
    virtual
    override(ERC20PausableUpgradeable, ERC20Upgradeable)
{
    USD0XTokenStorageV0 storage $ = _usd0XTokenStorageV0();
    if ($.isBlacklisted[from] || $.isBlacklisted[to]) {
        revert Blacklisted();
    }
    super._update(from, to, amount);
}
```

Remediation:

The explicit blacklist check in **_depositUSD0X()** is unnecessary and can be safely removed to reduce code duplication and maintain consistency with the token's internal access control logic.

USL3-3 | _depositUSD0X() Should Include USDC Blacklist Check

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/swapperEngine/SwapperEngine.sol#L192-L218

Description:

The **SwapperEngine._depositUSD0X()** function is responsible for creating a swapper order and transferring USD0X tokens from the sender to the contract:

```
function _depositUSD0X(
    SwapperEngineStorageV0 storage $,
    uint256 amountToDeposit,
    uint256 minimumUsd0xPrice
) internal {
    if (amountToDeposit < $.minimumUSD0XAmountProvided) {
        revert AmountTooLow();
    }

    if ($.usd0xToken.isBlacklisted(msg.sender)) {
        revert NotAuthorized();
    }

    uint256 orderId = $.nextOrderId++;
    $.orders[orderId] = SwapperOrder({
        requester: msg.sender,
        tokenAmount: amountToDeposit,
        active: true,
        minimumUsd0xPrice: minimumUsd0xPrice
    });

    $.usd0xToken.safeTransferFrom(msg.sender, address(this), amountToDeposit);

    emit Deposit(msg.sender, orderId, amountToDeposit);
}
```

Currently, the function only checks whether the sender is blacklisted on USD0X. However, if an attacker blacklisted on USDC creates an order with favorable terms, any user attempting to

match this order will experience a revert during the USDC transfer:

```
$.usdcToken.safeTransferFrom(msg.sender, order.requester, usdcAmountForOrderWithoutFees);
```

This leads to wasted gas for the taker, even though the attacker's order is otherwise valid.

Remediation:

Add a check for the sender's USDC blacklist status within **_depositUSD0X()** to prevent blacklisted addresses from creating orders and causing failed matches.

USL3-4 | minimumUsd0xPrice Should Reflect the USD0X–USDC Exchange Rate

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/swapperEngine/SwapperEngine.sol#L192-L196

Description:

The **SwapperEngine._depositUSD0X()** function creates an order for selling USD0X in exchange for USDC. Each order includes a parameter **minimumUsd0xPrice**, which represents the minimum acceptable price of USD0X in USD.

However, this parameter may not accurately enforce the user's intended constraint. Since the purpose of the order is to swap USD0X for USDC, not for USD directly, the relevant threshold should depend on the USD0X–USDC exchange rate rather than the USD value.

```
function _depositUSD0X(
    SwapperEngineStorageV0 storage $,
    uint256 amountToDeposit,
    uint256 minimumUsd0xPrice
) internal {
```

Remediation:

Consider replacing **minimumUsd0xPrice** with a parameter representing the minimum acceptable USD0X/USDC price, ensuring the slippage check aligns with the actual token pair being traded.

hexens x USUAL