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 focuses on the USD0++ Upgrade & Redemption Token for the Usual protocol. The upgrade introduces the bASSET0 and rt-ASSET0 tokens, representing the next iteration of the existing USD0++ design - a zero-coupon, bond-like instrument. Users enter the bond by deconstructing their principal into yield-bearing (bASSET0) and redemption (rt-ASSET0) tokens at a 1:1 ratio.

Our security review spanned one week and included a thorough evaluation of all contracts updated or added in this release.

No major vulnerabilities were discovered. We identified six informational-level issues during the assessment.

All findings were either fixed or acknowledged by the development team and subsequently verified by our auditors.

After the remediation process, we conclude that the protocol's security posture and code quality have been significantly improved 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/core-protocol/tree/b533cbbb6af61e97d8971d9625cc73fc32758d94

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

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

📌 Commit: `a92f83a15c698347751ceb206e3cfa2c8ec51a5c`

- **Changelog**

17 November2025	Audit start
24 November 2025	Initial report
26 November 2025	Revision received
03 December 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.

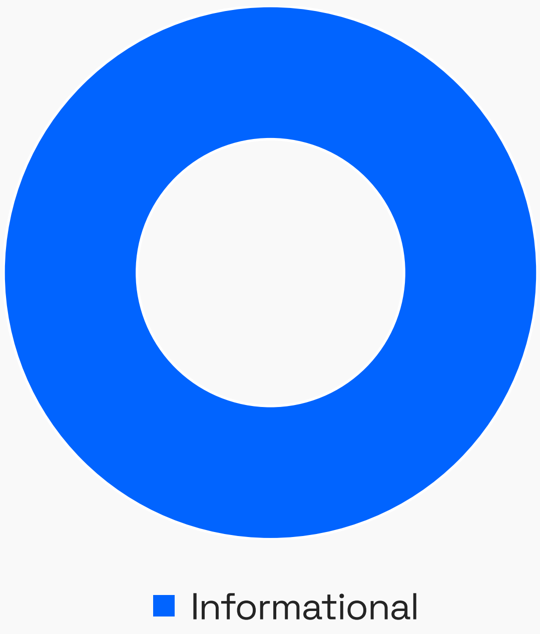
- ## Issue Symbolic Codes

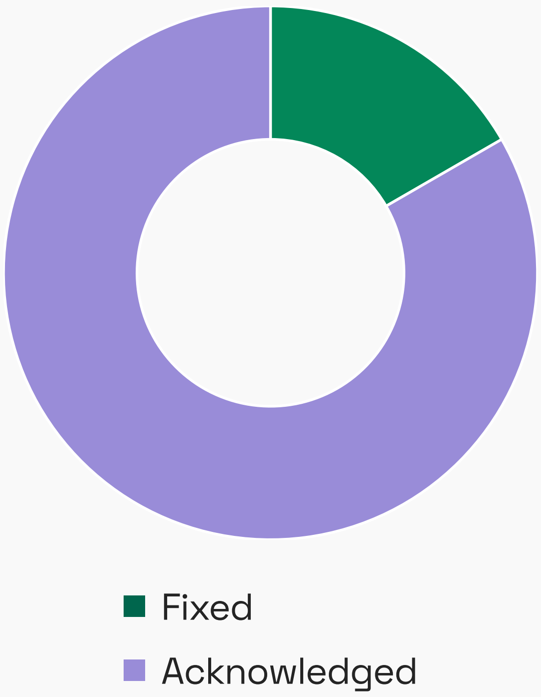
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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	0
🟩 Low	0
🟦 Informational	6
Total:	**6**





6. Weaknesses

This section contains the list of discovered weaknesses.

USL4-1 | Inconsistent Use of Modifiers in Mint Functions

Acknowledged

Severity:	Informational	Probability:	Rare	Impact:	Informational

Path:

`src/token/Usd0PP.sol#L197-L199`

Description:

The **mint(uint256 amountUsd0)** and **mint(uint256 amountUsd0, address bAssetRecipient, address rAssetRecipient)** functions are both protected with the **nonReentrant** and **whenNotPaused** modifiers.

However, the **mintWithPermit()** function does not include these modifiers, resulting in inconsistent security and pause protection across the different minting pathways.

```
function mintWithPermit(uint256 amountUsd0, uint256 deadline, uint8 v, bytes32 r, bytes32 s)
    external
{
```

Remediation:

Add the **whenNotPaused** and **nonReentrant** modifier to **mintWithPermit()** to ensure consistent pause behavior across all minting pathways.

```
    function mintWithPermit(uint256 amountUsd0, uint256 deadline, uint8 v, bytes32 r, bytes32 s)
        external
++      nonReentrant
++      whenNotPaused
    {
```

USL4-2 | Redundant Permit Call in unlockUSD0ppWithUsualWithPermit

Fixed ✓

Severity:	Informational	Probability:	Rare	Impact:	Informational

Path:

`src/token/Usd0PP.sol#L392-L402`

Description:

In the **Usd0PP.unlockUSD0ppWithUsualWithPermit()** function, lines 392–402 invoke **permit()** to grant the contract an allowance to spend the caller's **Usd0PP** tokens:

```
function unlockUSD0ppWithUsualWithPermit(
    uint256 usd0ppAmount,
    uint256 maxUsualAmount,
    PermitApproval calldata usualApproval,
    PermitApproval calldata usd0ppApproval
) external {
    Usd0PPStorageV0 storage $ = _usd0ppStorageV0();

    // Execute the USUAL permit
    try IERC20Permit(address($.usual))
        .permit(
            msg.sender,
            address(this),
            maxUsualAmount,
            usualApproval.deadline,
            usualApproval.v,
            usualApproval.r,
            usualApproval.s
        ) {}
        catch {} // solhint-disable-line no-empty-blocks

    // Execute the bUSD0 permit
    try IERC20Permit(address(this))
        .permit(
            msg.sender,
            address(this),
            usd0ppAmount,
            usd0ppApproval.deadline,
            usd0ppApproval.v,
            usd0ppApproval.r,
            usd0ppApproval.s
          ) {}
          catch {} // solhint-disable-line no-empty-blocks

      // Call the standard unlock function
      unlockUSD0ppWithUsual(usd0ppAmount, maxUsualAmount);
  }
```

This second permit call is unnecessary. The underlying **unlockUSD0ppWithUsual()** function burns the caller's **Usd0PP** using the internal **_burn()** method, which deducts tokens directly from **msg.sender** and does not require any allowance. As a result, granting approval for **Usd0PP** via **permit()** provides no functional benefit and introduces redundant logic.

Remediation:

Consider removing the second permit call in function **unlockUSD0ppWithUsualWithPermit()**.

USL4-3 | Use SCALAR_ONE Instead of Hard-Coded 1e18

Acknowledged

Severity:	Informational	Probability:	Rare	Impact:	Informational

Path:

src/token/Usd0PP.sol#L301-L303
src/token/Usd0PP.sol#L327

Description:

Several parts of the contract **Usd0PP** use the literal value **1e18** directly instead of the defined constant **SCALAR_ONE**. This leads to inconsistencies and makes future refactoring or scalar-related changes more error-prone.

Two instances where **1e18** is used directly include:

1. Comparison of **newFloorPrice** inside **updateFloorPrice()**

```
if (newFloorPrice > 1e18) {
    revert FloorPriceTooHigh();
}
```

2. Calculation of **usd0Amount** inside **unlockUsd0ppFloorPrice()**

```
uint256 usd0Amount = Math.mulDiv(usd0ppAmount, $.floorPrice, 1e18, Math.Rounding.Floor);
```

Using the hard-coded literal breaks the intended abstraction of the scalar system and may cause inconsistencies if **SCALAR_ONE** is ever modified or if the scaling logic needs to evolve.

Remediation:

Consider replacing all occurrences of **1e18** with the predefined constant **SCALAR_ONE** to ensure consistency.

USL4-4 | Missing Bond-Start Check in _deconstruct() Function

Acknowledged

Severity:	Informational	Probability:	Rare	Impact:	Informational

Path:

src/token/Usd0PP.sol#L568-L592

Description:

In the previous contract version, the **mint()** function explicitly validated that the bond period had already begun before allowing any minting. The logic looked as follows:

```
function mint(uint256 amountUsd0) public nonReentrant whenNotPaused {
    Usd0PPStorageV0 storage $ = _usd0ppStorageV0();

    // revert if the bond period isn't started
    if (block.timestamp < $.bondStart) {
        revert BondNotStarted();
    }

    // revert if the bond period is finished
    if (block.timestamp >= $.bondStart + BOND_DURATION_FOUR_YEAR) {
        revert BondFinished();
    }

    // get the collateral token for the bond
    $.usd0.safeTransferFrom(msg.sender, address(this), amountUsd0);

    // mint the bond for the sender
    _mint(msg.sender, amountUsd0);
}
```

This implementation correctly reverted when users attempted to mint before the bond period had started.

In the updated version, **mint()** delegates its core logic to the internal **_deconstruct()** function. However, **_deconstruct()** no longer includes the check to ensure that the current timestamp is past **bondStart**, only validating whether the bond period has already ended:

```
function _deconstruct(uint256 amountUsd0, address bAssetRecipient, address rAssetRecipient)
    internal
{
    Usd0PPStorageV0 storage $ = _usd0ppStorageV0();

    if (amountUsd0 == 0) {
        revert AmountIsZero();
    }

    // revert if the bond period is finished
    if (block.timestamp >= $.bondStart + BOND_DURATION_FOUR_YEAR) {
        revert BondFinished();
    }

    // get the collateral token for the bond
    $.usd0.safeTransferFrom(msg.sender, address(this), amountUsd0);

    // mint the bond token for the specified recipient
    _mint(bAssetRecipient, amountUsd0);

    // mint the redemption token for the specified recipient
    $.rtusd0.mint(rAssetRecipient, amountUsd0);

    emit Deconstructed(msg.sender, amountUsd0, bAssetRecipient, rAssetRecipient);
}
```

As a result, users are now able to mint before the bond period begins - behavior that differs from the previous design and likely violates the intended lifecycle of the bond.

Remediation:

Consider reintroducing the missing check:

```
if (block.timestamp < $.bondStart) {
    revert BondNotStarted();
}
```

inside **_deconstruct()** function.

USL4-5 | Redundant whenNotPaused Modifier Usage in Usd0PP Contract

Acknowledged

Severity:	Informational	Probability:	Rare	Impact:	Informational

Path:

```
src/token/Usd0PP.sol#mint()
src/token/Usd0PP.sol#mintWithPermit()
src/token/Usd0PP.sol#unwrap()
src/token/Usd0PP.sol#unwrapWithCap()
src/token/Usd0PP.sol#unwrapPegMaintainer()
src/token/Usd0PP.sol#unlockUsd0ppFloorPrice()
src/token/Usd0PP.sol#unlockUSD0ppWithUsual()
src/token/Usd0PP.sol#reconstruct()
```

Description:

In the **Usd0PP** contract, several public functions such as **unwrap()** and **mint()** explicitly include the **whenNotPaused** modifier. However, each of these functions ultimately calls **_mint()** or **_burn()**, both of which invoke the internal **_update()** function.

The **_update()** function in **ERC20PausableUpgradeable** already applies the whenNotPaused modifier:

```
// contract ERC20PausableUpgradeable

function _update(address from, address to, uint256 value)
    internal
    virtual
    override
    whenNotPaused
{
    super._update(from, to, value);
}
```

As a result, the pause check is executed twice - once at the beginning of the public function and again during the ERC-20 token state update. This redundancy unnecessarily increases gas consumption without adding any additional safety.

Remediation:

Consider removing the **whenNotPaused** modifier from the public functions that rely on **_mint()** or **_burn()**, since the underlying logic already enforces the pause constraint.

USL4-6 | pause() and totalBondTimes() Can Be Marked external Instead of public

Acknowledged

Severity:	Informational	Probability:	Rare	Impact:	Informational

Path:

src/token/Usd0PP.sol#L166
src/token/Usd0PP.sol#L485

Description:

In the **Usd0PP** contract, the functions **pause()** and **totalBondTimes()** are currently declared as **public**. However, neither of these functions is called internally within the contract.

```
function pause() public {
```

```
function totalBondTimes() public pure returns (uint256) {
```

Remediation:

Because they are only intended for external use, it is more efficient to mark them as **external** instead of **public**. Declaring them as **external** can reduce gas costs when invoked and more accurately reflects their intended usage.

hexens x USUAL