Usual Pegasus

Security Review

Cantina Managed review by:

Xmxanuel, Lead Security Researcher
Phaze, Security Researcher

January 31, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Usual is a Stablecoin DeFi protocol that redistributes control and redefines value sharing. It empowers users by aligning their interests with the platform's success.

From Dec 10th to Dec 12th the Cantina team conducted a review of usual-pegasus on commit hash 7d00272e. The team identified a total of **15** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	1	0	0
Low Risk	7	0	0
Gas Optimizations	0	0	0
Informational	7	0	0
Total	**15**	**0**	**0**

3 Findings

3.1 Medium Risk

3.1.1 `DistributionModule._calculateTotalSupply` should use `$.sUsdeVault.totalAssets()` instead `$.sUsdeVault.totalSupply()`

Severity: Medium Risk

Context: DistributionModule.sol#L988

Description: The `_calculateTotalSupply` function in the `DistributionModule` should return the total supply of `USD0++` tokens. The `USD0++` supply consists of the `USD0++.totalSupply` plus USD0 tokens which got invested into a `sUsdeVault`.

When funds are removed from the `sUsdeVault` vault they are transferred back into the `USD0++` bond as USD0 tokens. Therefore, the value of the current locked in the `vault` should be considered part of the `USD0++.totalSupply`. The underlying asset of the `sUsdeVault` is `Usde`. These `Usde` would be exchanged to `USD0` and added to the USD0++ bond in case of an exit.

For the correct calculation of additional USD0++ tokens in the bond would be to take the amount `Usde` tokens in the `vault` and multiply it with the current price of `Usde`. However, this calculation is done incorrectly by calling `$.sUsdeVault.totalSupply()` which would return the `ERC4646` total shares instead of total amount of `Usde` tokens.

```
- uint256 vaultValueUSD = $.sUsdeVault.totalSupply();
+ uint256 vaultValueUSD = $.sUsdeVault.totalAssets();
```

If the `sUsdeVault` is implemented correctly according to the ERC4646 specification the `totalAssets` function must be called to get the total amount of the underlying asset.

Implications: This error results in an incorrect `supplyFactor` (`S_t`) which would impact the `mintingRate` and leads to an incorrect distribution of Usual tokens. Specifically, the `supplyFactor` would not account for the yield generated by the `sUsdeVault` if we assume an initial `1:1` ratio between shares and assets in the vault.

Recommendation: Use `$.sUsdeVault.totalAssets();` int the `_calculateTotalSupply` function.

Usual: Deprecated.

3.2 Low Risk

3.2.1 Price timeout period exceeds Chainlink heartbeat significantly

Severity: Low Risk

Context: constants.sol#L143

Description: The contract sets a price timeout period of 7 days:

```
uint256 constant PRICE_TIMEOUT = 7 days;
```

However, the Chainlink price feed used by the contract has a heartbeat of 1 day, meaning it's guaranteed to update at least once every 24 hours. The current 7-day timeout is significantly longer than necessary and could allow the use of stale prices in case of feed issues, potentially leading to incorrect valuations or manipulated transactions.

Recommendation: Consider reducing the `PRICE_TIMEOUT` constant to a value closer to but slightly above the Chainlink heartbeat to ensure timely detection of stale prices while allowing for some network delays:

```
- uint256 constant PRICE_TIMEOUT = 7 days;
+ uint256 constant PRICE_TIMEOUT = 2 days;
```

This provides sufficient buffer for network delays while ensuring that significantly stale prices are not used for critical operations.

Usual: Deprecated.

3.2.2 Deployment scripts use invalid hardcoded placeholder addresses

Severity: Low Risk

Context: Contracts.s.sol#L430

Description: The deployment scripts use invalid hardcoded addresses (`0x01`) for initializing the distribution module:

```
abi.encodeCall(DistributionModule.initializeV1, (address(0x01), address(0x01)))
```

This is problematic because:

1. The contract's `initializeV1()` function explicitly checks for and reverts on zero addresses.
2. A valid Chainlink price feed address constant `SUSDE_CHAINLINK_PRICE_ORACLE` exists but isn't being used.
3. No valid vault address constant exists, but a placeholder value won't work in production.

Recommendation: Consider updating the deployment script to use the correct Chainlink price feed address and require a valid vault address as a deployment parameter.

Usual: Deprecated.

3.2.3 Function signature mismatch between interface and implementation

Severity: Low Risk

Context: IUsualX.sol#L8

Description: There is a mismatch between the `sweepFees()` function signature in the `IUsualX` interface and its implementation in the `UsualX` contract:

```
// IUsualX interface
function sweepFees(address collector) external;

// UsualX implementation
function sweepFees() external nonReentrant {
```

This inconsistency can cause integration issues for contracts that attempt to interact with `UsualX` through its interface. Additionally, since `UsualX` does not explicitly inherit from `IUsualX`, the mismatch is not caught at compile time.

Recommendation: Consider making `UsualX` explicitly inherit from its interface to catch such mismatches at compile time:

```
contract UsualX is IUsualX {
    // ... rest of the contract
}
```

Usual: Deprecated.

3.2.4 Inconsistent error handling in price oracle interactions

Severity: Low Risk

Context: DistributionModule.sol#L999-L1023

Description: The `_getSafeUsdePrice()` function handles oracle failures inconsistently. When the oracle is operational but returns potentially invalid data (negative price, zero roundId, etc.), the function reverts. However, when the oracle call itself fails (oracle is down), the function returns 0:

```
try $.sUsdePriceFeed.latestRoundData() returns (...) {
    if (answer <= 0) revert InvalidPrice();
    if (roundId == 0) revert InvalidPrice();
    if (updatedAt == 0) revert InvalidPrice();
    if (answeredInRound < roundId) revert StalePrice();
    if (block.timestamp - updatedAt > PRICE_TIMEOUT) revert StalePrice();

    price = uint256(answer);
} catch {
    //NOTE: if the oracle is down return 0
    price = 0;
}
```

This creates an inconsistency where some error conditions halt operations (through reverts) while others silently continue with a zero price, which could be just as dangerous as an invalid price.

Recommendation: Consider adopting a consistent error handling strategy based on the risk tolerance of the system:

1. Use zero as a safe fallback.
2. Any issue in retrieving the price should halt operations.

Usual: Deprecated.

3.2.5 Unwrap caps can only decrease

Severity: Low Risk

Context: Usd0PP.sol#L474

Description: The `unwrapWithCap()` function implements a one-way decremental cap system - caps can only decrease whenever tokens are unwrapped, with no mechanism to restore them:

```
function unwrapWithCap(uint256 amount) external nonReentrant whenNotPaused {
    // ... access control checks ...

    if (amount > $.unwrapCaps[msg.sender]) {
        revert AmountTooBigForCap();
    }

    $.unwrapCaps[msg.sender] -= amount;

    // ... unwrap logic ...
}
```

This design has the following implications:

1. Once a cap is exhausted, it remains at zero permanently unless manually reset by admin.
2. Users could purposefully exhaust the cap for integrating protocol with the `USD0PP_CAPPED_UNWRAP_ROLE` when wrapping and unwrapping USD0PP through the cap is permitted.

Recommendation: Consider documenting and making integrating protocols aware of this behavior, or consider allowing the unwrap cap to be replenished.

Usual: Acknowledged, the cap is set by the companion role

3.2.6 No explicit management of new storage slots during contract upgrades in `USD0++`

Severity: Low Risk

Context: Usd0PP.sol#L118-L127

Description: The `USD0++` contract is already deployed on Mainnet and will receive new storage variables in the upgrade to the new version. In current upgrade version these are only append at the end of the `Usd0PPStorageV0` struct.

```
  /// @custom:storage-location erc7201:Usd0PP.storage.v0
  struct Usd0PPStorageV0 {
      /// The start time of the bond period.
      uint256 bondStart;
      /// The address of the registry contract.
      IRegistryContract registryContract;
      /// The address of the registry access contract.
      IRegistryAccess registryAccess;
      /// The USD0 token.
      IERC20 usd0;
      uint256 bondEarlyUnlockStart;
      uint256 bondEarlyUnlockEnd;
      mapping(address => uint256) bondEarlyUnlockAllowedAmount;
      mapping(address => bool) bondEarlyUnlockDisabled;
      /// The current floor price for unlocking USD0++ to USD0 (18 decimal places)
      uint256 floorPrice;
+     /// The USUAL token
+     IUsual usual;
+     /// Tracks daily USD0++ inflows
+     mapping(uint256 => uint256) dailyUsd0ppInflows;
+     /// Tracks daily USD0++ outflows
+     mapping(uint256 => uint256) dailyUsd0ppOutflows;
+     /// USUAL distributed per USD0++ per day (18 decimal places)
+     uint256 usualDistributionPerUsd0pp;
+     /// The percentage of burned USUAL that goes to the treasury (basis points)
+     uint256 treasuryAllocationRate;
+     /// Daily redemption target rate (basis points of total supply)
+     uint256 targetRedemptionRate;
+     /// Duration cost adjustment factor in days
+     uint256 durationCostFactor;
+     /// Mapping of addresses to their unwrap cap
+     mapping(address => uint256) unwrapCaps;
  }
```

Appending the new variables at the end of the `struct` will not result in storage collisions. However, to avoid potential errors such changes should be more explicit in the codebase.

Recommendation: Introduce a new `Usd0PPStorageV1` which will have the old variables in the beginning and then clearly indicate with comments, where the new variables are introduced. Replace all usages of the `Usd0PPStorageV0` with `Usd0PPStorageV1` in the upgrade.

Alternatively, consider a logically separate of the newly needed storage struct slots. Like introducing `Usd0PPDistributionStorage` for the new variables to completely avoid potential storage collision mistakes.

Usual: Not acknowledged, to us this is more of an informational nature.

3.2.7 `USD0PP.setUnwrapCap` can be frontrunned by a vault to exceed the cap

Severity: Low Risk

Context: Usd0PP.sol#L447

Description: The `cap` of a `vault` for an `unwrapWithCap` operation can only be changed by calling `setUnwrapCap` with a new absolute `cap` value. This means if the `cap` amount should be increased or decreased, it could be front-run by a vault transaction.

Example: Suppose the cap is currently `100k` and you want to decrease it to `90k`. Before you call `setUnwrapCap`, the vault could front-run your transaction by calling `unwrapWithCap` using the current `100k` cap, thereby redeeming tokens accordingly. After the `setUnwrapCap` transaction sets the cap to `90k`, the vault could call `unwrapWithCap` again, ending up receiving a total of `190k`.

This attack is similar to a classic `ERC20 approve` front-running issue.

Recommendation: Depending on the trust assumptions for the vault, consider using an `int256` delta amount instead of an absolute new cap value. The delta amount can be positive or negative, reflecting an increase or decrease, respectively.

Usual: Acknowledged.

3.3 Informational

3.3.1 Redundant access control mechanism for unwrapping

Severity: Informational

Context: Usd0PP.sol#L463-L468

Description: The unwrapping functionality implements two layers of access control that are redundant. Currently, users must both:

1. Be assigned the `USD0PP_CAPPED_UNWRAP_ROLE`.
2. Have a non-zero unwrap cap set by an account with `UNWRAP_CAP_ALLOCATOR_ROLE`.

Since the unwrap cap effectively controls who can unwrap and by how much, the additional role check is unnecessary and adds complexity without providing additional security benefits.

Recommendation: Consider removing the `USD0PP_CAPPED_UNWRAP_ROLE` check since the unwrap cap mechanism already provides sufficient access control:

```
- $.registryAccess.onlyMatchingRole(USD0PP_CAPPED_UNWRAP_ROLE);

  // Check cap is set
  if ($.unwrapCaps[msg.sender] == 0) {
      revert UnwrapCapNotSet();
  }
```

Usual: Not acknowledged, since the `USD0PP_CAPPED_UNWRAP_ROLE` allows for a on-off setting without having to set/unset the value.

3.3.2 Incorrect error message when Usual amount exceeds maximum

Severity: Informational

Context: Usd0PP.sol#L648-L650

Description: The function reverts with `UsualAmountTooLow()` when the `requiredUsual` exceeds `maxUsualAmount`. However, this error message is misleading as it indicates the opposite of the actual condition being checked. The code is checking if the required amount is too high, not too low.

Recommendation: Update the error message to correctly reflect the condition being checked:

```
  if (requiredUsual > maxUsualAmount) {
-     revert UsualAmountTooLow();
+     revert UsualAmountTooHigh();
  }
```

Usual: Acknowledged.

3.3.3 Documentation and code comment inconsistencies

Severity: Informational

Context: constants.sol#L179

Description: There are discrepancies between the code comments, actual values, and whitepaper documentation:

1. The `INITIAL_USUAL_BURN_TARGET_REDEMPTION_RATE` is set to 30 basis points (0.3%), but the whitepaper states it should be 0.2%.
2. The `RATE0` constant has a misleading comment:

   ```
   uint256 constant RATE0 = 400; // 4.03% in basis points
   ```

   The value 400 basis points equals 4.00%, not 4.03%. Either the constant should be 403 to match the comment, or the comment should be updated to "4.00% in basis points".

Recommendation: Consider updating either the code or documentation to ensure consistency:

1. For `INITIAL_USUAL_BURN_TARGET_REDEMPTION_RATE`:

- Either update the constant to match the whitepaper:

```
uint256 constant INITIAL_USUAL_BURN_TARGET_REDEMPTION_RATE = 20; // 0.2% in basis points
```

- Or update the whitepaper to reflect the actual implementation of 0.3%.

2. For `RATE0`, either:

```
uint256 constant RATE0 = 403; // 4.03% in basis points
```

or

```
uint256 constant RATE0 = 400; // 4.00% in basis points
```

Usual: Deprecated.

3.3.4 Clearly indicate that the `USD0PP.targetRedemptionRate` is on a weekly and not daily basis

Severity: Informational

Context: Usd0PP.sol#L128

Description: The `targetRepemptionRate` is applied weekly in the `USD0++` contract. Most likely through refactoring the comment still incorrectly refers to them as `daily`.

```
/// Daily redemption target rate (basis points of total supply)
```

Recommendation: Rename the variable to `weeklyTargetRepemptionRate` and fix the incorrect comment to avoid all potential future errors when updating the `rate`.

3.3.5 Security guidelines for `signers` on mult-sigs like the `treasury` in `Usual`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The following are recommendations for signers on critical multi-sigs like the `treasury` in the Usual protocol which holds a high amount of RWA tokens.

Recommendation: -Use a Dedicated Signer Address: Always use a new, separate `signer` address for multi-sig wallets unless it belongs to an existing actor in the `Usual` protocol. Like another multi-sig, etc...

- Test Signer Control with an Initial Transaction: Before using a `signer` address in a multi-sig wallet, perform a small transaction to confirm control over the address or request a specific signature from the `signer`.
- Avoid using deployment addresses as Signers: Using an `address` for both contract deployments and as a `signer` in a multi-sig wallet can unnecessarily increase risk, as deployment keys are often stored in less secure environments to allow efficient deployments.
- Avoid Defi Interactions with Signers: Signers in a multi-sig wallet should avoid direct interactions with DeFi protocols. Such activities expose the signer's private key to potential vulnerabilities, including signing incorrect messages or malicious transactions.

3.3.6 If condition in `YieldBearingVault._updateYield` could be more specific

Severity: Informational

Context: YieldBearingVault.sol#L145-L147

Description: The `_updateYield` function has the following if condition:

```
$.lastUpdateTime = Math.min(block.timestamp, $.periodFinish);
// if we are at the end of the period, deactivate yield
if ($.lastUpdateTime >= $.periodFinish) {
  $.isActive = false;
}
```

Logically, the `lastUpdateTime` can never be > than `$periodFinish`.

- If `block.timestamp > $.periodFinish`, `$.lastUpdateTime` will be equal to `$.periodFinish`.
- If `block.timestamp <= $.periodFinish`, `$.lastUpdateTime` will be equal to `block.timestamp`.

This guarantees that `$.lastUpdateTime` can never exceed `$.periodFinish`.

Recommendation: The if condition could be changed to be more specific

```
if ($.lastUpdateTime == $.periodFinish) {
  $.isActive = false;
}
```

Usual: Deprecated.

3.3.7 Early revert in `USD0PP.unlockUSD0ppWithUsual` if `usd0ppAmount=0` would simplify `calculateRequiredUsual` logic

Severity: Informational

Context: Usd0PP.sol#L644

Description: Adding an early revert in `unlockUSD0ppWithUsual` if `usd0ppAmount=0`. Would simplify the logic `calculateRequiredUsual` functions. The return value of `calculateNetOutflows` would be always > 0 and the `calculateAdjustmentFactor` would not have to handle the additional `netOutflows==0` case.

Recommendation: Add the following early revert to the function.

```
if (usd0ppAmount == 0) {
  revert Usd0PPAmountIsZero();
 }
```

Usual: Given input is 0 and output should be 0 not an error.