Usual Pegasus: Phase 1
Security Review

Cantina Managed review by:
Deadroses.xyz, Lead Security Researcher
Phaze, Security Researcher
Chinmay Farkya, Associate Security Researcher

November 1, 2024

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.

USD0 is a USUAL native stablecoin with real-time transparency of reserves, fully collateralized by US Treasury Bills. This eliminates fractional reserve risks and protects against the bankruptcy risks of fiat-backed stablecoins.

$USD0 can be locked into $USD0++, a liquid 4-year bond backed 1:1, offering users the alpha-yield distributed as points and ensuring at least the native yield of their collateral. This provides enhanced stability and attractive returns for holders.

From Oct 8th to Oct 22nd the Cantina team conducted a review of pegasus on commit hash 747f595f. The team identified a total of **10** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 4
- Gas Optimizations: 2
- Informational: 3

3 Findings

3.1 Medium Risk

3.1.1 PAR mechanism mistakenly transfers USD0 tokens to `AirdropDistribution` contract instead of treasury

Severity: Medium Risk

Context: Usd0PP.sol#L347-L402

Description: The USD0++ token's Parity Arbitrage Right (PAR) mechanism contains a mistake in its implementation. When activated, excess USD0 tokens are incorrectly sent to the `AirdropDistribution` contract instead of the intended DAO treasury, potentially making funds inaccessible.

The USD0++ token includes a PAR mechanism designed to stabilize its price relative to USD0 when it falls below parity on the open market. This mechanism can be activated when more USD0++ tokens exist on the market than USD0 tokens And the price ratio of USD0:USD0++ is below 1. When triggered, the mechanism unwraps USD0++ for USD0 and exchanges it on the market to support the USD0++ price. Any excess USD0 received from this operation is supposed to be sent to the DAO's treasury.

However, the current implementation incorrectly forwards this excess amount to the `AirdropDistribution` contract. This contract is not designed to hold tokens and lacks methods for transferring them out, potentially leading to locked funds.

Impact: The impact of this vulnerability can potentially be significant, as funds are effectively locked in the `AirdropDistribution` contract. Yet, as the contract is upgradeable the impact can be lowered to medium. Further, addressing this in an emergency upgrade carries its own risks and complications.

Likelihood: The likelihood of this issue occurring is high when the PAR mechanism is activated. The PAR mechanism itself is, however, not intended to be used frequently lowering the likelihood of the vulnerability occurring to medium.

Recommendation: To address this issue, modify the `triggerPARMechanismCurvepool` function in the `Usd0PP` contract to transfer the USD0 tokens to the correct address:

```
  usd0.safeTransfer(
-     $.registryContract.getContract(CONTRACT_AIRDROP_DISTRIBUTION), gainedUSD0AmountPAR
+     $.registryContract.getContract(CONTRACT_TREASURY), gainedUSD0AmountPAR
  );
```

Usual: Fixed in commit ae6ea94e.

Cantina Managed: Fixed.

3.2 Low Risk

3.2.1 Timestamp parameters can be desynchronized from the vesting schedule

Severity: Low Risk

Context: AirdropDistribution.sol#L146-L180, AirdropTaxCollector.sol#L126-L145, Usd0PP.sol#L181-L195

Description: The `AirdropTaxCollector` contract's initializer function requires a start date parameter, which is stored and used in calculations such as the claim tax amounts and the claiming period windows. This start date is also used in the `AirdropDistribution` contract.

Currently, deployment scripts initialize this date using a predefined constant: `AIRDROP_INITIAL_START_TIME = 1_734_004_800` (Dec 12 2024 12:00:00 GMT+0000). Even though this value is stored in the contract's storage, it is not modifiable. Changing this date after deployment could potentially impact other contracts that assume it remains constant.

The `AirdropDistribution` contract's calculation for available airdrop tokens assumes a fixed start date, as the vesting schedule is also fixed. Additionally, in the `Usd0PP`, the early unlock period for bonds is defined to last 19 days from the vesting start date, ending at `END_OF_EARLY_UNLOCK_PERIOD = 1_735_686_000` (31st Dec 2024 23:00:00 GMT+0000).

This predefined timeline eliminates the need for configurable parameters that might be overlooked during setup or become desynchronized if one date changes. Using constants for these dates can improve the clarity and robustness of contracts. Alternatively, computing all dates relative to a single, centrally defined start date would make time adjustments easier if necessary.

Recommendation:

1. Simplify the `AirdropTaxCollector` contract by hardcoding the known start and end dates (replacing `$.startDate`).
2. In the `AirdropDistribution` contract, use the fixed, known start date instead of `$.startTime`.
3. Define the bond early unlock period based on the known start date.

Usual: Partially fixed in commit c7383554. We've fixed the first two points by hardcoding the start date in the `AirdropTaxCollector` and using the fixed start date in the `AirdropDistribution`. The third point remains unchanged as the early unlock period is managed by an admin-only function, and we've scheduled its start.

3.2.2 Penalties can be applied to past and future months

Severity: Low Risk

Context: AirdropDistribution.sol#L192-L215, AirdropDistribution.sol#L365-L394

Description: The airdrop system for users in the "top 80" group includes a penalty mechanism that can reduce their token allocation if they violate certain terms. These penalties can be applied monthly, up to 100% of the allocation for each month. While users can claim their airdrop tokens monthly, the current system allows for an unclear and potentially unfair application of penalties:

1. Retroactive penalties: Penalties can be applied to past months, potentially wiping out unclaimed balances from periods when the user was compliant.
2. Future penalties: The system allows penalties to be applied to future months, immediately voiding upcoming claims.
3. Tax payment loophole: Users can still pay taxes to claim some earnings, but this triggers an additional month of penalty, further reducing their allocation.

This system creates the potential for an unjust treatment and can lead to confusion around the terms and the airdrop rewards.

Proof of Concept:

- Scenario 1: Retroactive penalties. Alice hasn't claimed her tokens for 3 months:

Month	1	2	3
Claimable	100	200	300
Penalty	0	0	0

  After an action that displeases the team, all previous months are penalized:

Month	1	2	3
Claimable	0	0	0
Penalty	100	200	300

  Alice loses the 300 tokens she should have been able to claim.

- Scenario 2: Future penalties. Alice claimed all earnings up to month 3:

Month	1	2	3	4	5	6
Claimable	0	0	0	100	200	300
Penalty	0	0	0	0	0	0

Team applies future penalties:

Month	1	2	3	4	5	6
Claimable	0	0	0	0	0	0
Penalty	0	0	0	100	200	300

Alice loses all future claims instantly.

- Scenario 3: Tax payment loophole. After applying the future penalties in scenario 2, Alice can exercise the option to pay the tax for month 4. Thereby she can redeem a leftover claim of 200 tokens bypassing the 300 token penalty.

Month	1	2	3	4	5	6
Claimable	0	0	0	0	100	200
Penalty	0	0	0	100	0	0

Recommendation: Limit the penalty application to the upcoming month only. For example, behavior from month 0 to 1 should only affect the claim for month 1. And behavior from month 5 to 6 should only affect month 6. Applying these changes would make the penalty system more predictable and fair for users.

Usual: Partially fixed in c7383554. We've addressed point 1 by removing retroactive penalties, as setting penalties for past months was unnecessary. However, penalties for future months remain since we plan to apply them just before the next month starts and we do not want to restrict it further. For point 3, the penalty is still deducted even if the user pays taxes, as it applies to the following month when a penalty is planned.

3.2.3 Front-running penalties is possible by claiming early

Severity: Low Risk

Context: AirdropDistribution.sol#L316-L343, AirdropDistribution.sol#L365-L394

Description: The airdrop system for "top 80" users contains a vulnerability that allows users to potentially evade penalties by quickly claiming their tokens upon seeing an incoming penalty transaction.

The airdrop system includes a penalty mechanism for users who violate certain terms. However, the current implementation allows for a scenario where users can avoid these penalties:

1. Users can monitor the mempool for incoming penalty transactions.
2. Upon detecting a penalty, users can quickly pay the tax for early claims.
3. Users can then immediately claim their outstanding airdrop tokens.
4. The incoming penalty becomes ineffective as there are no more unclaimed tokens to penalize.

This option could undermine the penalty system and allow users avoid consequences for violating the airdrop terms.

Impact: Exploiting this vulnerability could give certain users an unfair advantage. However, the potential gain is diminished by the amount of tax that must be paid upon claiming tokens early.

Likelihood: The likelihood of this vulnerability being exploited is considered low, as it requires monitoring the mempool and technical knowledge to execute the required transactions quickly to avoid the penalty.

Proof of Concept:

1. Alice sees a transaction in the mempool to penalize her airdrop allocation.
2. Alice calls `payTaxAmount`, paying the tax for an early claim.
3. Alice immediately calls `claim` to claim her outstanding allocation.
4. The incoming penalty does not apply to Alice as she has claimed all outstanding tokens.

Recommendation: To address this issue, consider introducing a small time delay window (e.g., 1 hour) between paying the tax and allowing the next claim. Alternatively, given the low likelihood, the project could simply accept the risk.

Usual: Acknowledged. We didn't fix this issue, as we accept the risk. Paying the tax benefits the protocol, so the impact is minimal. Implementing an anti-frontrun mechanism would be too complex for this scenario and isn't necessary given the low severity of the finding.

3.2.4 [REDACTED]

Severity: Low Risk

Description: [REDACTED]

Usual: Not acknowledged.

(Redacted until 31th of December).

3.3 Gas Optimization

3.3.1 `Claim()` can be optimized for cases when merkle root is not set

Severity: Gas Optimization

Context: AirdropDistribution.sol#L316-L330

Description: `Claim()` function verifies that the merkle proof shared by claimer is valid and if it evaluates to the airdrop merkle root set by admin. But it does not check that the merkle root actually exists in the `AirdropDistribution` contract.

In case the `merkle root == bytes(0)`, the likelihood of this getting exploited is very low, but checking this first in the `claim()` flow can save some gas by reverting early.

Recommendation: Add a check `require($.merkleRoot != 0);` at the start of the `claim()` function for better code clarity, this also saves some gas by reverting early in case its not set.

Usual: Not fixed. We didn't address this finding, as the merkle root will be set before the airdrop start time. Since the likelihood of exploitation is very low, we opted not to add an extra check for an empty merkle root in the `claim()` function.

3.3.2 Full precision math is not strictly required

Severity: Gas Optimization

Context: AirdropDistribution.sol#L192-L215

Description: The codebase uses full precision math whenever fractions are computed. While this can prevent overflow issues and allows for specifying a rounding direction, in most cases it is not required adding unnecessary computational overhead.

When the claimable token amount is computed in the `AirdropDistribution` contract, the total available amount is multiplied by the number of months passed and divided by the number of months in total.

```
claimableAmount = totalAmount.mulDiv(monthsPassed, AIRDROP_VESTING_DURATION_IN_MONTHS);
```

This calculation could overflow if the total amount exceeded $(2^{256} - 1)/6$. This would require a user to receive a token allocation exceeding $192986815395526992372618308347813179755449974442734273399 09 \cdot 10^{18}$ tokens, an unlikely high amount.

When the penalty amount is computed, first the monthly amount is computed, rounding up.

```
uint256 oneSixthAmount =
    totalAmount.mulDiv(1, AIRDROP_VESTING_DURATION_IN_MONTHS, Math.Rounding.Ceil);
```

As the total amount is multiplied by one and then divided, the full precision computation is not required. A rounding direction is given, which rounds against the user. However, the effective maximum difference will be less than `6 wei` tokens, a negligible amount. This amount is further capped, such that the penalty

cannot exceed the claimable amount. Rounding in favor of the user has no further implications for the solvency of the protocol in this case.

When accumulating the total penalty, the monthly available amount is multiplied by the fractional penalty amount.

```
uint256 monthlyPenalty =
    oneSixthAmount.mulDiv($.penaltyPercentageByMonth[account][i], BASIS_POINT_BASE);
```

This computation could only overflow if the total allocation for a user exceeded $(2^{256} - 1) \cdot \frac{6}{10000} = 6947525354238971725414259100521274471196199079938433842 3 \cdot 10^{18}$ tokens.

If more precision is desired, the multiplication can be performed first before the division. The tax fee calculation in the `AirdropTaxCollector` contract also uses full precision math.

```
uint256 taxFee = $.maxChargeableTax.mulDiv(claimingTimeLeft, AIRDROP_CLAIMING_PERIOD_LENGTH);
claimTaxAmount = claimerUsd0PPBalance.mulDiv(taxFee, BASIS_POINT_BASE);
```

For similar reasons as stated before, these calculations do not require full precision math and do not specify a rounding direction, making it safe to replace these with the default math operations.

Recommendation: Consider only applying full precision fractional calculations when it is necessary and use default math calculations in order to reduce complexity and save gas.

- `AirdropDistribution`:

````
    ```diff
    - claimableAmount = totalAmount.mulDiv(monthsPassed, AIRDROP_VESTING_DURATION_IN_MONTHS);
    + claimableAmount = totalAmount * monthsPassed / AIRDROP_VESTING_DURATION_IN_MONTHS;
    ```

    ```diff
    - uint256 monthlyPenalty =
    -     oneSixthAmount.mulDiv($.penaltyPercentageByMonth[account][i], BASIS_POINT_BASE);
    + uint256 monthlyPenalty =
    +     totalAmount * $.penaltyPercentageByMonth[account][i] / BASIS_POINT_BASE / 6;
    ```
````

- `AirdropTaxCollector`:

````
    ```diff
    - uint256 taxFee = $.maxChargeableTax.mulDiv(claimingTimeLeft, AIRDROP_CLAIMING_PERIOD_LENGTH);
    - claimTaxAmount = claimerUsd0PPBalance.mulDiv(taxFee, BASIS_POINT_BASE);
    + uint256 taxFee = $.maxChargeableTax * claimingTimeLeft / AIRDROP_CLAIMING_PERIOD_LENGTH;
    + claimTaxAmount = claimerUsd0PPBalance * taxFee / BASIS_POINT_BASE;
    ```

    More precision can be achieved by first multiplying all factors and then dividing.

    ```diff
    - uint256 taxFee = $.maxChargeableTax.mulDiv(claimingTimeLeft, AIRDROP_CLAIMING_PERIOD_LENGTH);
    - claimTaxAmount = claimerUsd0PPBalance.mulDiv(taxFee, BASIS_POINT_BASE);
    + claimTaxAmount = claimerUsd0PPBalance * $.maxChargeableTax * claimingTimeLeft /
    ↪  AIRDROP_CLAIMING_PERIOD_LENGTH / BASIS_POINT_BASE;
    ```
````

Usual: Not fixed. We didn't make changes to this finding, as the current use of `mulDiv` is consistent across the contract. Additionally, removing the ceiling rounding would leave some dust amounts, which we want to avoid.

3.4 Informational

3.4.1 Dead logic and constants

Severity: Informational

Context: Usd0PP.sol#L537-L553

Description: `_createUsd0PPCheck` function in `USD0PP.sol` is dead code, not used anywhere in the codebase. Also, there are some constants that are unused currently. These are roles that have been granted but no actions are associated with these roles in current code :

- `SWAPPER_ENGINE`
- `CONTRACT_ORACLE_USUAL`
- `GAMMA_PRECISION`
- `BOND_START_DATE`
- `ADMIN`

Recommendation: Remove the `_createUsd0PPCheck` function, and the constants (roles) if they are not required.

Usual: Fixed in c7383554. The `_createUsd0PPCheck` function is used in `src/mock/token/Usd0PPHarness.sol::Usd0PPHa` But to simplify the Usd0PP.sol contract, we've moved the code directly into `Usd0PPHarness.sol`. We have also removed the unused constants.

3.4.2 Documentation errors

Severity: Informational

Context: IUsd0PP.sol#L34, IUsd0PP.sol#L98

Description: At several places in the codebase, comments need to be corrected.

1. `IUSD0PP.sol::mintWithPermit()` $\Rightarrow$ should be `Transfers collateral Usd0 tokens and mints Usd0PP bonds.`
2. `IUSD0PP.sol::setBondEarlyUnlockDisabled()` $\Rightarrow$ should be `Only callable by airdrop tax collector contract.`

Recommendation: Apply the mentioned changes.

Usual: Fixed in commit c7383554.

3.4.3 Wrong error used in `initialize()` function of `AirdropDistribution` contract

Severity: Informational

Context: AirdropDistribution.sol#L163-L165

Description: In the initializing logic of `AirdropDistribution.sol`, if the airdrop start time is not in the correct range (ie. before `FIRST_AIRDROP_VESTING_CLAIMING_DATE`), then the initialization reverts with an error `AmountTooBig()`.

This error is not relevant to this check.

Recommendation: Use error `InvalidClaimingPeriodStartDate()` in place of `AmountTooBig()`.

Usual: Fixed in commit c7383554. We now use the `InvalidClaimingPeriodStartDate()` error instead of `AmountTooBig()`.