CANTINA

Pegasus
Competition

August 13, 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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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 Jun 18th to Jun 28th Cantina hosted a competition based on Pegasus. The participants identified a total of **5** issues in the following risk categories:

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

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Updating the `registryAccess` doesn't affect the `AbstractOracle`

Submitted by ZdravkoHr, also found by 0xffchain, Niroh and zigtur

Severity: Medium Risk

Context: AbstractOracle.sol#L94

Description: The `RegistryContract` is used to store global contracts used by the system in a mapping.

```
mapping(bytes32 => address) _contracts;
```

It also has a function to update this mapping:

```
function setContract(bytes32 name, address contractAddress) external {
    // ...
    $._contracts[name] = contractAddress;
    emit SetContract(name, contractAddress);
}
```

When `AbstractOracle` is initialized, the `registryAccess` is pulled from this mapping:

```
$.registryAccess = IRegistryAccess($.registryContract.getContract(CONTRACT_REGISTRY_ACCESS));
```

The problem arises when the `registryAccess` is updated in the `registryContract`. This update will not reflect the `AbstractOracle` contract and the oracle will continue using the old `registryAccess`.

Impact: High → The oracle will use a deprecated `registryAccess` contract which grants users access to critical functionalities, like `setMaxDepegThreshold`.

Likelihood: Medium → Requires `registryAccess` to be updated in the `registryContract`.

Recommendation: Add a function in the `AbstractOracle` which syncs the `registryAccess` with the `registryContract` one.

Usual: Acknowledged. In case of a change regarding the `registryAccess`, we already have an operational procedure in place that requires the re-initialization of the `AbstractOracle` too. Additionally, given that the `registryAccess` contract itself is a transparent proxy, this address wouldn't even change when the `registryAccess` is upgraded.

3.1.2 Blacklisted user can bypass the blacklist

Submitted by Silvermist, also found by jesjupyter and carrotsmuggler

Severity: Medium Risk

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

Description: If a user is blacklisted he should not be able to make any interactions with the protocol, however, there is a way to bypass the blacklist.

Let's take a look at `_swapRWAtoStbc` function:

1. RWA token is transferred from `msg.sender` to the contract.
2. The corresponding amount of usd0 is minted to the contract.
3. `swapUsd0` is executed and inside it usd0 is transferred from the `DaoCollateral` contract to `order.recipient` because `msg.sender` is the address of the `DaoCollateral` contract.

Both `mint` and `safeTransferFrom` check if the user is blacklisted, but since interactions with usd0 are done on behalf of the `DaoCollateral` contract, the check will pass even though the user initiating the transaction is blacklisted.

Proof of concept: Paste the following test inside `DaoCollateral.t.sol`:

```
function testUserCanBypassTheBlacklist() public {
    vm.prank(admin);
    stbcToken.blacklist(address(alice));

    uint256 amountInUsd0 = 42_000 * 1e18;
    uint256 amountToDeposit =
        _getUsdcAmountFromUsd0WadEquivalent(amountInUsd0, _getUsdcWadPrice());
    (
        RwaMock rwaToken,
        uint256 rwaAmount,
        uint256[] memory orderIdsToTake,
        Approval memory approval
    ) = setupSwapRWAToStbc_withDeposit(amountToDeposit);

    vm.prank(alice);
    daoCollateral.swapRWAtoStbc(address(rwaToken), rwaAmount, true, orderIdsToTake, approval);
    assertEq(IUSDC(address(USDC)).balanceOf(bob), 0);
    assertEq(IUSDC(address(USDC)).balanceOf(alice), amountToDeposit);

}
```

Recommendation: Check if the user is not blacklisted inside the `_swapRWAtoStbc`:

```
if (IUsd0(address($.usd0)).isBlacklisted(caller)) {
    revert NotAuthorized();
}
```

Usual: Acknowledged, but we disagree with the severity. The reasoning at the time was that our RWA Token providers have a stringent KYC process for their Token Allowlist, which makes a malicious RWA Provider who also needs to be blacklisted by the Protocol extremely unlikely. Furthermore, we are on par with USDC's Sanctions list, which would also be automatically enforced by the nature of having to receive USDC.

3.1.3 Counter Bank Run mechanism activation should be taken into account in `SwapperEngine` to protect users offering USDC

Submitted by ktl, also found by iamandreiski, zark and twcctop Exvul

Severity: Medium Risk

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

Description: When CBR is activated in `DaoCollateral` USD0 → RWA redemption value drops by the CBR coefficient and all operations on `DaoCollateral` are paused except for the redeem function. However, with the current implementation this has no effect on the `SwapperEngine`.

The assumption is the CBR would be activated when total value of the rwa tokens locked in the protocol is less than the total supply of USD0 tokens, or in extreme market conditions relating the USD0 peg or RWA price variations. Another assumption is that once CBR is activated, it would only be deactivated once the total value of the rwa tokens locked in the protocol is more than the total supply either by direct transfer or by price appreciation.

USDC offerers in `SwapperEngine` are referred to as makers and USD0 swappers are referred to as takers.

The activation of the CBR would reduce the value of USD0 by `~CBR_coefficient`, makers might not want to exchange their stablecoin in such conditions as it would be almost certainly unfavorable to them: when CBR is activated their USDC is exchanged for USD0 1-1 wheras the underlying RWA that could be redeemed from `DaoCollateral` just decreased.

This would not be prevented by the depeg protection in the oracle, as `SwapperEngine` queries for the price of USDC, wich is expected to remain stable regardless of RWA/USD0 state and price volatility.

Makers have no way to specify that they only wish to exchange their stable coin only when the CBR is off (aka the `DaoCollateral` is healthy), and takers will be incentivized to exchange all their USD0 holdings to USDC until CBR is reactivated.

Impact: From a taker's perspective, once CBR is activated the only logical thing to do is exchange their USD0 holdings to any other stablecoin because USD0 would already have depegged on DEX exchanges or

at an imminent risk of depeg. This is because `SwapperEngine` considers that USD0's price is $1 == 1USD0 at all times, even if it depegs from all markets.

From a maker's perspective, swapping while CBR is activated is disadvantegeous for two main reasons:

1. Obviously the USD0 underlying they can redeem in `DaoCollateral` has less value.
2. They might have logic that depends on their orders in `SwapperEngine` getting fullfilled to execute subsequent trades. Example: a maker operating a bot that would listen for events emitted when an order is fullfilled then redeems USD0 for RWA in `DaoCollateral` and executes some arbitrage opportunity elsewhere. This might end up with unexpected results (potentially a loss) in case CBR is on.

My assessment is that the first issue is more severe because the only way for makers who do not want to perform swaps in such market conditions is to constantly monitor the mempool for calls to `DaoCollateral.activateCBR()` and cancel all their orders, this would be very impractical for the makers as it would involve frontrunning any tx fulfilling their orders.

The second issue could be easily avoided by checking whether CBR is on before redeeming but it might be too late to cancel the order and the maker might end up with USD0 that he wasn't planning on holding.

In my opinion the impact is low because although the current behaviour might result in losses for makers it will not exceed CBR_coefficient and the protocol incurs no loss.

Likelihood: Once CBR is activated all USD0 holders are incentivized to exchange all their holding through the `SwapperEngine` as it would be cheaper than most DEXes because of fees and USD0 might have already depegged on DEXes.

This will essentially create a bank run on the `SwapperEngine`. The likelihood is therefore high in my opinion, as market participants are expected to behave rationally.

Proof of concept: This is a minimal proof of concept to demonstrate that `SwapperEngine` swaps are not paused when CBR is activated in `DaoCollateral`.

1. Copy the following functions into `packages/solidity/test/swapperEngine/swapperEngine.t.sol`.
2. run with `forge test --match-path test/swapperEngine/swapperEngine.t.sol --match-test 'test_poc_*'`.

```
function activateCBR() internal {
    vm.startPrank(admin);

    // activate cbr
    daoCollateral.activateCBR(1);

    assertTrue(daoCollateral.isCBROn());
}

function test_poc_swapper_engine() public {
    uint256 amountUsd0ToProvideInWad = 1000 * 1e18; // 1000 USD0
    uint256[] memory orderIdsToTake = new uint256[](1);
    orderIdsToTake[0] = 1;

    // Setup: Create a USDC order for Alice
    uint256 amountToDeposit =
        _getUsdcAmountFromUsd0WadEquivalent(amountUsd0ToProvideInWad, _getUsdcWadPrice());
    vm.startPrank(alice);
    deal(address(USDC), alice, amountToDeposit);
    IUSDC(address(USDC)).approve(address(swapperEngine), amountToDeposit);
    swapperEngine.depositUSDC(amountToDeposit);
    vm.stopPrank();

    // Execute: Bob swaps USD0 for USDC
    vm.startPrank(bob);
    deal(address(stbcToken), bob, amountUsd0ToProvideInWad);
    IERC20(address(stbcToken)).approve(address(swapperEngine), amountUsd0ToProvideInWad);
    uint256 unmatched =
        swapperEngine.swapUsd0(bob, amountUsd0ToProvideInWad, orderIdsToTake, false);
    vm.stopPrank();

    // Assert: Check the unmatched amount and token balances
    assertEq(unmatched, 0);
    assertEq(IUSDC(address(USDC)).balanceOf(bob), amountToDeposit);
    assertEq(IERC20(address(stbcToken)).balanceOf(alice), amountUsd0ToProvideInWad);
```

```
}

function test_poc_swapper_engine_with_cbr_on() public {
    activateCBR();
    test_poc_swapper_engine();
}
```

Recommendation: On the operational level this could be solved by an admin atomically pausing `SwapperEngine` and activating CBR on `DaoCollateral`. On the protocol level there are two solutions:

1. When CBR is activated, also pause `SwapperEngine` except for order cancellation, this would require adding the pauser role to `DaoCollateral`.
2. Allow makers to specify wether they want to go ahead with swaps if CBR is on, being false by default to protect makers.

One way would be to introduce the following modifications:

```
  struct UsdcOrder {
      address requester;
      uint256 tokenAmount;
      bool active;
+     bool swapWhenCBROn;
  }
```

Then in `SwapperEngine._provideUsd0ReceiveUSDC()`:

```
  // assuming SwapperEngine now has a reference to DaoCollateral
+ isCBROn = DaoCollateral.isCBROn();
  for (
      uint256 i;
      (...)
+ if (order.active && (isCBRon ? offer.swapWhenCBROn : true))
```

Makers would be pass `swapWhenCBROn` when provididing USDC. In my opinion the second solution is better because it would have the same effect as the first one but it gives makers more flexibility.

Both solutions require introducing some shared state between `SwapperEngine` and `DaoCollateral` to check for CBR state, wich isn't present in the current code. My recommendation would be to move the CBR state to a common contract instead of adding a reference to `DaoCollateral` in `SwapperEngine` (or vice-versa/both).

Usual: Acknowledged. We already have had a remediation for that risk planned in a contract upgrade, which required additional independent oracles from third parties based on secondary markets.