MakerDAO: USDC Vault

Security Review

Cantina Managed review by:
Christoph Michel, Lead Security Researcher
M4rio.eth, Security Researcher

February 26, 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

The Maker Protocol, also known as the Multi-Collateral Dai (MCD) system, allows users to generate Dai (a decentralized, unbiased, collateral-backed cryptocurrency soft-pegged to the US Dollar) by leveraging collateral assets approved by the Maker Governance, which is the community organized and operated process of managing the various aspects of the Maker Protocol.

From Feb 6th to Feb 11th the Cantina team conducted a review of spark-vaults on commit hash 5fb9d321. The directories in scope for this review were:

- `src`
- `deploy`

The team identified **1** issue:

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

The Cantina team reviewed MakerDAO's `spark-vaults` holistically on commit hash b0ea091a and concluded that all the issues were addressed and no new vulnerabilities were identified.

3 Findings

3.1 Informational

3.1.1 Wrong test assertion in `testMaxWithdraw`

Severity: Informational

Context: UsdcVaultL2-integration.t.sol#L585-L587, UsdcVault-integration.t.sol#L713-L715

Description: The `testMaxWithdraw(uint256 depositAmount, uint256 warp)` fuzz test in `UsdcVaultL2-integration.t.sol` expects different revert messages when attempting to withdraw `maxWithdraw + 1`, based on how `maxWithdraw` relates to the USDC pocket balance:

```
if (maxWithdraw < usdc.balanceOf(pocket)) {
    vm.expectRevert("SafeERC20/transfer-from-failed");
} else {
    vm.expectRevert("SafeERC20/transfer-failed");
}
token.withdraw(maxWithdraw + 1, address(this), address(this));
```

The `withdraw` performs a `PSM3.swapExactOut(sUSDS -> USDC, exactOut)` which internally first performs a `sUSDS.transferFrom()` followed by the `usdc.transfer()` payout. Note that all `withdraw` calls with `maxWithdraw+1` will revert already in the `sUSDS.transferFrom()` call with the first "SafeERC20/transfer-from-failed" error:

1. If `maxWithdraw <= usdcPocketBalance - 1`, we are attempting to withdraw at most `usdcPocketBalance`. So it would never revert with a USDC transfer error as we're doing an `exactSwapOut(susds -> usdc, exactOut=usdcPocketBalance)` which always transfers out the `exactOut` and not more.
2. The interesting case is `maxWithdraw == usdcPocketBalance` when we attempt to withdraw `usdcPocketBalance + 1`.
    1. If other users have a `token` balance such that the `token`'s sUSDS balance is higher, the `sUSDS.transferFrom` would indeed pass and we would revert at the `usdc.transfer` with `SafeERC20/transfer-failed`.
    2. However, the user is the only `token` balance holder in the test and in this case withdrawing `usdcPocketBalance + 1` already leads to withdrawing more than the `token`'s total sUSDS balance, resulting in an early revert.
3. `maxWithdraw > usdcPocketBalance` can never happen as `maxWithdraw` is itself computed as the minimum of `usdcPocketBalance` and another value.

If the fuzz tests run long enough and the `maxWithdraw == usdcPocketBalance` condition is hit, it will revert with a false positive counter-example for `testMaxWithdraw`. (See `testFuzz_maxWithdraw` test case below.)

> **Note:** The same is true for the `maxWithdraw` L1 tests like `testMaxWithdraw(uint256 depositAmount, uint256 tout, uint256 warp)`: They always revert in the `sUSDS` `transferFrom` already with "SUsds/insufficient-balance" when trying to withdraw more than the USDC pocket balance as there's a single token holder that only holds shares equivalent to at most the USDC pocket balance. However, the revert error case distinction done for the L1 / L2 `maxRedeem` tests is correct and necessary.

Recommendation: Consider changing:

```
- if (maxWithdraw < usdc.balanceOf(pocket)) {
-     vm.expectRevert("SafeERC20/transfer-from-failed");
- } else {
-     vm.expectRevert("SafeERC20/transfer-failed");
- }
+ vm.expectRevert("SafeERC20/transfer-from-failed");
```

Furthermore, if a function is defined as a `minimum` (like `maxWithdraw` and `maxRedeem`) we highly recommend writing an explicit test for the case when both `min` arguments are equal as it's the most critical point and non-guided fuzzers will usually never hit it as this issue showcases. Consider adding fuzz tests similar to these:

```
// tests displayed are only for L2, similar ones could be added for L1
function testFuzz_maxWithdraw(uint256 userShares, uint256 warp) public {
```

```
    // fuzz maxWithdraw
    userShares = userShares % (1e12 * 1e18);
    warp %= 365 days;
    vm.warp(block.timestamp + warp);

    deal(address(susds), address(token), userShares); // token escrows susds for redemptions
    deal(address(token), address(this), userShares); // users gets same token shares as susds shares

    // given userShares, compute the usdcPocketBalance equivalent to these user shares so we end up with the
    ↪  same values for the `min` in maxWithdraw
    uint256 usdcUserBalance = (userShares / 1e12) * RateProviderLike(rateProvider).getConversionRate() / 1e27;
    uint256 usdcPocketBalance = usdcUserBalance;

    // maxWithdraw := min(usdcUserBalance, usdcPocketBalance)
    deal(address(usdc), address(pocket), usdcPocketBalance);

    uint256 maxWithdraw = token.maxWithdraw(address(this));
    assertEq(maxWithdraw, usdcPocketBalance, "maxWithdraw mismatch");

    vm.expectRevert("SafeERC20/transfer-from-failed");
    token.withdraw(maxWithdraw + 1, address(this), address(this));
    if (maxWithdraw == 0) vm.expectRevert("PSM3/invalid-amountOut");
    token.withdraw(maxWithdraw, address(this), address(this));
    assertEq(token.maxWithdraw(address(this)), 0);
}

function testFuzz_maxRedeem(uint256 usdcPocketBalance, uint256 warp) public {
    // fuzz maxRedeem
    usdcPocketBalance = usdcPocketBalance % (1e12 * 1e6);
    warp %= 365 days;
    vm.warp(block.timestamp + warp);

    deal(address(usdc), address(pocket), usdcPocketBalance);

    // as defined in maxRedeem
    // given usdcPocketBalance, compute the user shares equivalent so we end up with the same values for the
    ↪  `min` in maxRedeem
    uint256 sharesPocketBalance = PsmLike(psm).previewSwapExactIn(address(usdc), address(susds),
    ↪  usdc.balanceOf(PsmLike(psm).pocket()));
    vm.assume(sharesPocketBalance > 0);
    uint256 wantedUserBalance = sharesPocketBalance;

    // maxRedeem := min(balanceOf(user), sharesPocketBalance)
    deal(address(susds), address(token), wantedUserBalance); // token escrows susds for redemptions
    deal(address(token), address(this), wantedUserBalance); // users gets same token shares as susds

    uint256 maxRedeem = token.maxRedeem(address(this));
    assertEq(maxRedeem, wantedUserBalance);

    token.redeem(maxRedeem, address(this), address(this));
    assertLt(token.maxRedeem(address(this)), 2 * 10**12);
    assertLt(usdc.balanceOf(address(pocket)), 2 * 1e6);
}
```

MakerDAO: Solved in commits 750f070b and b0ea091a.

Cantina Managed: Verified.