CANTINA

MakerDAO: XChain DSR Oracle

Security Review

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

September 9, 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

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.

On Aug 20th the Cantina team conducted a review of xchain-dsr-oracle on commit hash 000e4fd0.

The Cantina team reviewed MakerDAO's xchain-dsr-oracle changes holistically on commit hash cbd09d87 and determined that all issues were resolved and no new issues were identified.

The team identified a total of **8** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 2
- Gas Optimizations: 0
- Informational: 5

3 Findings

3.1 Medium Risk

3.1.1 Out-of-sync `DSRAuthOracle`'s pot data leads to conversion factor jumps

Severity: Medium Risk

Context: DSRAuthOracle.sol#L68, DSROracleBase.sol#L49

Description: The `DSRAuthOracle` uses the cached pot data to extrapolate the conversion rate (`chi`) in functions like `getConversionRate`. This cached pot data is updated through a cross-chain message, at least once whenever the `dsr` is updated on ETH. The extrapolated conversion rates can become out of sync with the actual `pot.chi()` and when new pot data is set in `DSRAuthOracle` the conversion rate will experience sudden jumps (both up and down are possible).

It can become out of sync if wrong pot data is submitted or pot data updates are not received in chronological order. It also happens naturally as there is a time delay between updating the `pot` on ETH and the message being received on the L2 through `DSRAuthOracle.setPotData`. Imagine the following timeline and states:

1. Both ETH's `pot` and L2's `DSRAuthOracle`'s pot state is in sync and set as `(rho_0, dsr_0, chi_0)`.
2. At time `rho_1` on ETH: `pot.drip()` is called and a new DSR is filed via `pot.file(dsr)`. This updates the state to `(rho_1, dsr_1, chi_1)`.
3. Immediately, at time `rho_1` on ETH, a cross-chain message is sent to L2 to update the state on the L2's oracle.
4. At time `tau`, the message is relayed on the L2 and `DSRAuthOracle.setPotData()` also updates its state to `(rho_1, dsr_1, chi_1)`. The conversion factor is now extrapolated with the new state.

On L2, the conversion factors were computed as:

- Before the update (`t < tau`): `chi_0 * dsr_0^(now - rho_0)`.
- After the update (`t >= tau`): `chi_1 * dsr_1^(now - rho_1)`.

During the time period `[rho_1, tau]` the pot data is out of sync and at the time of update `tau` the conversion factor on the L2 jumps. The difference (`after - before`) can be computed as:

```
chi_1 * (dsr_1^(tau - rho_1) - dsr_0^(tau - rho_1))
```

The new conversion factor increases when `dsr_1 > dsr_0` (decreases when `dsr_1 < dsr_0`) and the cross-chain update delay `tau - rho_1` amplifies this difference.

As the conversion rate can be used for pricing assets like L2 `sDAI` (see `PSM3` contract), the price jumps can be sandwiched and profited from. The profit per $ investment is the growth in `chi` which can be computed as `(dsr_1^(tau - rho_1) / dsr_0^(tau - rho_1)) - 1`.

See the proof of concept below that compute the profit per $. Essentially, it is always profitable because there are no swap fees, the profit (and therefore the LPs' loss) is just bounded by the number of sDAI tokens in the PSM3.

Proof of concept: The following script computes the profit percentage per $ spent by swapping into sDAI in the PSM3 before the new pot data is set, and then swapping back out of sDAI.

Some example values for profit / $ spent:

- DSR APR: 5% → 10%, delay = 1 day: 0.0138%
- DSR APR: 5% → 10%, delay = 7 days: 0.0968%
- DSR APR: 5% → 15%, delay = 1 days: 0.0277%
- DSR APR: 5% → 15%, delay = 7 days: 0.1937%

Assuming similar TVL as the ETH PSM is achieved and estimating 100M$ of sDAI in the PSM3 an attacker can steal 10,000$ in the first example, 193,700$ in the second example.

```python
def get_chi(oldChi, dsr, seconds_elapsed):
    return oldChi * dsr ** seconds_elapsed

def get_apr(dsr):
    return (dsr - 1) * 31536000

chi_0 = 1.5
dsr_old = 1.0000000016 # 5% APR
dsr_new = 1 + (dsr_old % 1.0) * 2 # 10% APR

print(f"DSR_old APR: {get_apr(dsr_old) * 100:.2f}%")
print(f"DSR_new APR: {get_apr(dsr_new) * 100:.2f}%")

# t_rho1 = rho_1 - rho_0
t_rho1 = 90 * 24 * 60 * 60 # DSR changes after 90 days (this value is irrelevant for profit)
# t_tau = tau - rho_1
t_tau = 7 * 24 * 60 * 60 # syncing takes this time
# chi on L2 before update runs for t_rho1 + t_tau seconds
chi_unsynced = get_chi(chi_0, dsr_old, t_rho1 + t_tau)
# chi on ETH runs with old dsr until dsr updates after t_0 seconds
chi_1 = get_chi(chi_0, dsr_old, t_rho1)
# then runs with new dsr t_tau seconds
chi_synced = get_chi(chi_1, dsr_new, t_tau)

print(f"chi_unsynced = {chi_unsynced:.6f}")
print(f"chi_synced = {chi_synced:.6f}")
chi_growth_percentage = (chi_synced - chi_unsynced) / chi_unsynced
print(f"chi growth = {chi_growth_percentage * 100:.4f}%")

# Note that the chi growth (jump percentage) is the profit that can be made.
# Verify
# 1. swap USD to sDAI on PSM3
usd_investment = 1_000_000
sDai_bought = usd_investment / chi_unsynced
# 2. setPotData runs on L2, synching chi
usd_received = sDai_bought * chi_synced
profit = usd_received - usd_investment
print(f"Profit per $: {profit / usd_investment * 100:.4f}%")

# Direct way to compute profit
# profit = (dsr_1^(tau - rho_1) / dsr_0^(tau - rho_1)) - 1
profit = (dsr_new ** t_tau / dsr_old ** t_tau) - 1
print(f"Profit per $: {profit * 100:.4f}%")
```

Recommendation: DSR updates on ETH should be communicated to other chains immediately to reduce the impact. If this cannot be guaranteed, consider smoothing out the conversion factor to its target value over time or reducing sDAI liquidity from PSM3 before a DSR update.

Maker: Acknowledged. We are aware of this and the idea is to have critical infrastructure keepers to perform this update ASAP when the DSR changes on mainnet. We expect that in practice the deviation will be small enough to be a negligible profit opportunity if not for arbitragers than at least the protocol compared to revenue.

It would be good to know profit opportunity % vs time delay on updating for something like a 5% change in DSR.

Also, out-of-order updates are not a problem as `DSRAuthOracle` enforces ordering at the application level via the `rho` value being only non-decreasing.

The numbers shown in the proof of concept look acceptable to me. We consider this oracle update critical infrastructure in the same way price oracle updates are. The plan is to propagate pot value changes within 20 minutes of the DSR changing. So 1 day is extreme. The average case will be more like 1% DSR change and ~20 min delay. Even in worst case scenarios losing 10k on 100m is not that big of a deal for something as infrequent as a DSR update.

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 Reduced range of `PotData` values on L2

Severity: Low Risk

Context: IDSROracle.sol#L12

Description: The `PotData` struct, that the `DSROracle`s use, has different types than the pot data on ETH. This is most important for the `chi()` variable defined as a `uint120` on L2 instead of a `uint256`.

With an initial value of `RAY = 1e27`, the `uint120` type is exceeded at 50% APY after 50 years (compared to 126 years when using 166 bits, the max bits before multiplying by `RAY` reverts in `_rpow`).

Recommendation: Consider extending the range of `PotData.chi` to a `uint256` (or the practical limit of `uint168`) to have the same effective ranges for the value on all chains.

Maker: Acknowledged. So the upper size limit is around `uint160` due to the chi multiplication at ray precision. If we extend the width by 40 bits, we will only approximately double the amount of years. 50% APY is unreasonable, but something like 7% on average is a more reasonable upper bound and we are talking more like 350 years at 120-bit size and ~700 at 160.

The larger point is that these accumulators always have human-ish level time frames due to the exponential nature regardless of the bit width and will need to be addressed somehow in the far future.

So I think these numbers are reasonable is bottom line.

Cantina Managed: Acknowledged.

3.2.2 `DSROracleForwarderBase._packMessage()` type-casts silently drop bits

Severity: Low Risk

Context: DSROracleForwarderBase.sol#L27-L29

Description: The `DSROracleForwarderBase._packMessage()` function reads three `uint256` fields from the `pot` and casts them to lower types.

```
dsr: uint96(pot.dsr()),
chi: uint120(pot.chi()),
rho: uint40(pot.rho())
```

If the fields exceed the types, the upper bits are silently dropped and wrong package data is sent cross-chain.

Recommendation: Consider using safe type-casts that check if the value fits in the new type and otherwise revert, protecting against sending wrong package data cross-chain.

Maker: Fixed in commit cbd09d87.

Cantina Managed: Fixed.

3.3 Informational

3.3.1 Improved require check in `getConversionRate`, `getConversionRateBinomialApprox` and `getConversionRateLinearApprox`

Severity: Informational

Context: DSROracleBase.sol#L52, DSROracleBase.sol#L70, DSROracleBase.sol#L112

Description: Within the `getConversionRate`, `getConversionRateBinomialApprox` and `getConversionRateLinearApprox` functions a require check is performed so that the `timestamp` parameter to be greater or equal with the `rho`. This can be improved from >= to > because the equality check is already checked in the above line. For example:

```
if (timestamp == rho) return d.chi;
require(timestamp >= rho, "DSROracleBase/invalid-timestamp");
```

Recommendation: Consider changing from >= to > in the mentioned functions.

Maker: Fixed in commit cbd09d87.

Cantina Managed: Fixed.

3.3.2 Obsolete `DEFAULT_ADMIN_ROLE` assignment to `DATA_PROVIDER_ROLE`

Severity: Informational

Context: DSRAuthOracle.sol#L23

Description: In the `AccessControl` from OpenZeppelin, the `DEFAULT_ADMIN_ROLE` is already the admin of any newly defined role.

- AccessControl.sol#L39:

```
/**
 * By default, the admin role for all roles is `DEFAULT_ADMIN_ROLE`, which means
 * that only accounts with this role will be able to grant or revoke other
 * roles. More complex role relationships can be created by using
 * {_setRoleAdmin}.
```

Recommendation: Remove the `_setRoleAdmin` call from the constructor.

Maker: Fixed in commit cbd09d87.

Cantina Managed: Fixed.

3.3.3 Improved checks in the `setPotData`

Severity: Informational

Context: DSRAuthOracle.sol#L34-L65

Description: Within the `setPotData` various checks are performed when updating the data:

- `nextData.dsr` to not be > `maxDSR`.
- `nextData.rho` to not be > `block.timestamp`.
- `nextData.dsr` to not be lower than `RAY`.

These checks are not performed if the data is initialized the first time.

Recommendation: Consider adding the checks before the following if statement:

```
if (_data.rho == 0) {
```

This will ensure that even the initial data will be checked against these violations as well.

Maker: Fixed in commit cbd09d87.

Cantina Managed: Fixed.

3.3.4 `DSROracle`'s pot data starts uninitialized

Severity: Informational

Context: DSROracleBase.sol#L48

Description: The `DSROracleBase`'s pot data `_data` field remains uninitialized until the first cross-chain `setPotData` message is received. During this time functions like `getConversionRate()` will return a conversion rate of `0` which could lead to errors for integrators or large jumps in conversion rates once the first pot data is received.

Recommendation: Consider reverting in the `getConversionRate(uint256 timestamp)`, `getConversionRateBinomialApprox(uint256 timestamp)` and `getConversionRateLinearApprox(uint256 timestamp)` in case `_data` is uninitialized.

Maker: Acknowledged. We are fine with leaving as-is. Sending the first update is part of the setup process, and we would consider it invalid to share anything before this is done. There is extra gas in the check for the hot path too.

Cantina Managed: Acknowledged.

3.3.5 `DSRBalancerRateProviderAdapter.getRate()` documentation improvement

Severity: Informational

Context: DSRBalancerRateProviderAdapter.sol#L23

Description: The `DSRBalancerRateProviderAdapter.getRate()` function's documentation states:

> @return The value of sDAI in terms of DAI.

Recommendation: Consider improving the documentation by stating that it uses a binomial approximation instead of the more accurate `_rpow` computation the sDAI function uses, and clarify that it is an approximation for `1e18` tokens of sDAI.

> @return The approximated value of 1e18 sDAI in terms of DAI.

Maker: Fixed in commit cbd09d87.

Cantina Managed: Fixed.