PUBLIC

Code Assessment

of the G-UNI LP Oracle Smart Contracts

November 17, 2021

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 4
3 Limitations and use of report 7
4 Terminology 8
5 Findings 9
6 Resolved Findings 10
7 Notes 12

1 Executive Summary

Dear Maker team,

First and foremost we would like to thank MakerDAO for giving us the opportunity to assess the current state of their G-UNI LP Oracle system. This document outlines the findings, limitations, and methodology of our assessment.

More extensive documentation, especially a short description/motivation of the underlying concept of the price feed and the related requirements on its dependencies (the oracles) would be helpful. Ideally such documentation is done for all intended GUnipools the price feed is to be used for. For details please refer to Missing Documentation. For this review we focused on the intended use as price feed for the USDC-DAI GUnipool, a pool with two stablecoins.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Specification Changed	1
Low-Severity Findings	2
• Code Corrected	1
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the G-UNI LP Oracle repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	18 October 2021	0d25676e0956fc3a5359431c708106547c4fef26	Initial Version
2	16 November 2021	f069d760976b6fb0c15c9a2453f0c0f1ff87ea21	After Intermediate Report

For the solidity smart contracts, the compiler version 0.6.12 was chosen. This version, although being deprecated, has been chosen explicitly to be consistent through all newly developed maker modules.

The file in scope for this review was: GUniLPOracle.sol. The main focus was on the internal `seek` function including a brief review of the callpath to `GUNI.getUnderlyingBalancesAtPrice()`. The internal workings of the GUNI functions however are not part of this review.

2.1.1 Excluded from scope

The GUni implementation itself and UniswapV3 are not part of this review. This includes all reused code from UniswapV3, GUNI and the function `sqrt()` from `ABDKMath64x64` library.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

GUniLPOracle is a specialized oracle in the Maker ecosystem that provides prices for the LP (liquidity provider) shares of GUNI pools. It determines the price of a GUni token based on the underlying tokens held in the UniswapV3 position at the current market rate of these tokens as returned by Maker oracle. GUNI works on top of Uniswap and serves as a generic wrapper of Uniswap V3 positions into ERC20 tokens with the goal to provide more flexibility to end-users that deposit or withdraw liquidity into Uniswap V3 pools.

On a high level, Uniswap V3 aims to utilize more efficiently the pool liquidity by allowing the LPs to choose the price range (`lowerTick` and `upperTick`) where their liquidity is made available. The rewards for an LP depend mostly on the trade volume on the price range that the liquidity has been allocated. This makes Uniswap V3 positions non-fungible. On the other side, GUNI is a module managed by Gelato Networks that tries to abstract the internals of the Uniswap V3 to end-users (LPs) and maximize their profits by allocating the liquidity continuously into optimal price ranges and investing the earned fees. In this setup, the LPs provide the liquidity into the GUNI pools, which deposit the liquidity into the Uniswap V3 and then mints the respective wrapped ERC20 tokens for the LP. Note that, the

minted tokens (shares) by GUNI represent a position in the Uniswap V3 pool, however, such tokens are typical ERC20 tokens, hence fungible (while Uniswap V3 positions are non-fungible).

The goal of GUniLPOracle is to price the LP shares of GUNI pools according to the value of the position they represent in the Uniswap V3 pool. To achieve this goal the GUniLPOracle interacts with other oracles in the Maker ecosystem that provide price information for the related tokens and the respective GUNI pool. For this to work, the GUNI should provide a function `getUnderlyingBalancesAtPrice(uint160 sqrtPriceX96)`, which forwards the call to the function `LiquidityAmounts.getAmountsForLiquidity()`. The core logic of the price calculation in GUniLPOracle is implemented in the function `seek()`. Similarly to other oracles of Maker, GUniLPOracle operates with two `Feed` variables `cur` and `nxt` which store the current price and the queued price respectively. The prices propagate through the system with 1 hour delay, therefore allowing `wards` to take measures in case the queued price `nxt` is set to an incorrect value.

GUniLPOracle provides the following functionalities:

- `stop()`: can be called only by authorized wards to stop the oracle.
- `start()`: can be called only by authorized wards to remove the stop flag `stopped = 0`.
- `step()`: can be called only by authorized wards to update the `hop` value (default 1 hour).
- `link()`: can be called only by authorized wards to update the oracle address for a token.
- `zzz()`: can be called by anyone and returns the timestamp of the last price update.
- `pass()`: can be called by anyone and returns `true` if enough time to compute the new price has passed since the last update.
- `poke()`: can be called by anyone and computes the new price of an LP share given that `pass()` returns `true`. The core logic of the price calculation is implemented in the function `seek()` which has `internal` visibility.
- `peek()`: can be called only by whitelisted addresses in the mapping `bud` and returns the current price and its validity.
- `peep()`: can be called only by whitelisted addresses in the mapping `bud` and returns the queued price (which will be set as current in the next call of `poke()`) and its validity.
- `read()`: can be called only by whitelisted addresses in the mapping `bud` and returns the current price as `bytes32`.
- `kiss()`: can be called only by authorized wards and sets a single (or an array of) address into the whitelist mapping `bud`.
- `diss()`: can be called only by authorized wards and removes a single (or an array of) address from the whitelist mapping `bud`.
- The standard authorization functions `rely()` and `deny()`.

GUniLPOracleFactory allows any user to deploy an GUniLPOracle by calling the function `build()` which takes as parameters:

- `address _owner`: the oracle calls `rely()` for this address.
- `address _src`: the address of GUNI pool whose LPs shares will be evaluated.
- `bytes32 _wat`: the label of the `_src` token.
- `address _orb0`: the address of oracle for token0.
- `address _orb1`: the address of oracle for token1.

2.2.1 Trust Model & Roles

Wards: Fully trusted to behave honestly and correctly at all times. They can set the parameters `hop`, `orb0`, `orb1`, can stop the oracle by calling `stop()` or resume it with `start()`, and add/remove whitelisted addresses to/from the mapping `bud`. We assume the `wards` monitor the price feed continuously and take measures in case `nxt` holds an incorrect price value before it propagates into the system.

GUNI pool: Fully trusted. Note that the implementation of a GUni pool is upgradable. Furthermore GUni pools have privileged roles `manager` that can modify the parameters of the pool arbitrarily. Both factors can impact the price feed significantly. We assume that the operators of GUni are fully trusted and they behave correctly. For the purpose of this audit GUni is expected to work as intended, including that it cannot be manipulated by flashloans. Finally, we assume the Uniswap V3 implements its functionalities according to the specification correctly.

External users: Untrusted. Can call the functions of GUniLPOracleFactory or GUniLPOracle with arbitrary parameters.

Oracles: Part of the Maker Ecosystem, fully trusted. Used to query the prices of the underlying tokens. These oracles return the **current** live rate without any delay.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Possible Gas Optimization for Mappings Acknowledged

5.1 Possible Gas Optimization for Mappings

Design Low Version 1 Acknowledged

Although the value for the mapping `isOracle` is of type `bool` which needs only 1 bit of storage, Solidity uses a word (256 bits) for each stored value and performs some additional operations when operating `bool` values (masking). Therefore, using `uint` instead of `bool` is slightly more efficient.

Acknowledged:

Maker acknowledged the issue.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Missing Documentation Specification Changed

Low-Severity Findings	1

- Unused Constant Variable Code Corrected

6.1 Missing Documentation

Design Medium Version 1 Specification Changed

The requirements about the oracles for the underlying tokens are not documented. In the supplied test file we see following oracles:

```
address constant USDC_ORACLE        = 0x77b68899b99b686F415d074278a9a16b336085A0;
address constant DAI_ORACLE         = 0x47c3dC029825Da43BE595E21fffD0b66FfcB7F6e;
address constant ETH_ORACLE         = 0x81FE72B5A8d1A857d176C3E7d5Bd2679A9B85763;
```

The oracles for USDC and DAI return the unit value of one. The ETH oracle is updated roughly once an hour hence the price returned is not live. For the proper working of the GUniLPOracle a live price feed is required, frequently updated and without a time delay. When `GUniLPOracle.seek()` is executed, the underlying price feeds must return live values.

Furthermore the underlying principle how the price is determined could be described more clearly in the Readme:

This price feed works by determining how many of token0 and token1 the underlying liquidity position in UniswapV3 held by the GUniPool has at the current price. This current price is solely determined by Maker oracles and independent of the current state of the UniswapV3 pool. The assumption is that

1. The Maker oracles for the underlying tokens return the current market rate
2. In general, e.g. outside flashloan scenarios, the UniswapV3 pool will be balanced at the current market rate. This means that the GUnipool tokens can be redeemed at this current market rate.

Hence such a GUnipool token collateral is priced based on its underlying tokens, independent of the state of the GUni/Uniswap V3 pool. The documentation may be expanded to explain and motivate this.

Specification changed:

Maker responded:

```
It was a mistake that the test was referring to the ETH/USD OSM. It should have
referenced the ETH/USD Medianizer to get a live price feed.
```

Furthermore the readme has been updated and now contains:

```
Underlying price oracles `orb0` and `orb1` should refer to either a Medianizer,
DSValue or some other `read()` compliant oracle. OSMs should not be used to
the double delay.
```

6.2 Unused Constant Variable

Design | Low | Version 1 | Code Corrected

The variable `WAD` is declared as constant and initialized to `10 ** 18`, however it's never used in the code.

Code corrected:

The unused constant has been removed.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Misleading Function Name `link`

Note Version 1

The function name `link(uint256 _id, address _orb)` is misleading as it gives the impression that the token `_id` is linked to the respective oracle initially by this function. However, this function only updates an existing *link* of the token with the respective oracle (initialized in the `constructor`).

7.2 `NewGUniLPOracle` Indexed Fields

Note Version 1

The event `NewGUniLPOracle` has two indexed parameters corresponding to the token addresses. In practice, it might be useful if the field `address owner` is indexed also, as it would allow users to easily filter oracles from a trusted owner.