PUBLIC

Code Assessment

of the Swaap v2 Euler Adapter Smart Contracts

8 Nov, 2024

Produced for

by

Contents

1 Executive Summary

Dear Swaap Labs,

Thank you for trusting us to help Swaap with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Swaap v2 Euler Adapter according to Scope to support you in forming an opinion on their security risks.

Swaap Labs implements `SwaapSafeguardOracle`, a price feed that integrates in the Euler Price Oracles system to allow using Liquidity Tokens of Swaap SafeguardPool as collateral in Euler vaults.

The most critical subjects covered in our audit are LP token price manipulation by unprivileged users, price manipulation by privileged users, and decimal precision in mathematical operations. For all aforementioned subjects the security is high.

The general subject covered by this audit is the integration of SafeguardPool LP tokens as collateral in Euler lending markets. Regarding this subject security is good, but as seen in Potentially significant underpricing in Some Scenarios, the price returned by the oracle is only a lower bound on the value of the LP tokens. Users whose positions are collateralized by `SafeguardPool` LP tokens should be aware of the pricing mechanism.

Since the price returned by the oracle is a lower bound, ChainSecurity reminds future users of `SwaapSafeguardOracle` that it can only be used to price the collateral of lending markets, and never the borrowable token.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

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	1
• Code Corrected	1
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	3
• Specification Changed	1
• Risk Accepted	2

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 Swaap v2 Euler Adapter 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	04 Oct 2024	5d0a49ad580ea32e260060ae10a719f84cf75fff	Initial Version
2	31 Oct 2024	979c8818423b81b28b67c83a2102a91dbb7c6f5f	Fixes

For the solidity smart contracts, the compiler version `0.8.23` was chosen.

Only `SwaapSafeGuardOracle.sol` was considered for the assessment.

2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section is considered out of scope. In particular, the overall system, the tests, deployment scripts, external dependencies, and configuration files are not part of this audit.

2.2 System Overview

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

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Swaap Labs is integrating Swaap SafeguardPool LP tokens into Euler's lending markets. The integration involves developing a custom adapter for the Euler Price Oracle (EPO) system. The custom adapter, `SwaapSafeGuardOracle`, will allow Euler Vaults to price the Swaap SafeguardPool LP tokens, so that they can be used as collateral.

2.2.1 Safeguard Pool

Swaap SafeguardPool LP tokens are ERC20 tokens representing liquidity provider (LP) positions in the Swaap SafeguardPool.

Swaap SafeGuardPools are liquidity pools that enable traders to swap between asset pairs while allowing LPs to earn trading fees. The pools are distinguished by two key features:

- **Off-chain quotation process:** price quotes are computed and signed off-chain and are then verified on-chain before execution.
- **Safeguards:** a set of safeguards is implemented to protect LPs against unfair or stale off-chain price quotes.

The price quotes are signed off-chain by the `signer` role. Every request to swap assets must contain a price quote signed by the `signer`. Price quotes are specific to a single swap request, and are not reusable.

The three main Safeguards are:

- **Max Price Deviation:** Prevents trades when the quoted price deviates too much from the on-chain oracle price in the direction unfavorable to the LP.
- **Max Performance Deviation:** Ensures the pool's performance does not deviate down beyond a set threshold compared to the last HODL updates.
- **Max Balance Deviation:** Limits the deviation between new pool balance and target reference balance.

2.2.2 Euler Price Oracle System

The Euler Price Oracle system is a modular system designed to fetch price data for tokens in Euler lending markets. It provides a collection of oracle adapters that implement the `IPriceOracle` interface, facilitating the integration of various price feeds and pricing mechanisms.

Key components of the Euler Price Oracle system include:

- `EulerRouter`: A registry contract that routes price queries to the appropriate oracle adapter based on the token pair.
- Adapters: Individual `IPriceOracle` contracts that implement specific pricing logic for different assets or asset pairs.

2.2.2.1 IPriceOracle Interface

The `IPriceOracle` interface defines the structure for adapters within the Euler Price Oracle system. It provides the following methods:

- `name()`: Returns the name of the oracle.
- `getQuote(base, quote, amountIn)`: Calculates the amount of quote token obtained for a given amountIn of base token, assuming no price spread.
- `getQuotes(base, quote, amountIn)`: Provides both `bidOutAmount` and `askOutAmount`, accounting for price spread when calculating the amounts.

2.2.2.2 Base Adapter

`BaseAdapter` is the abstract contract that serves as the base for all oracle adapters within the Euler Price Oracle system. It implements the `IPriceOracle` interface by:

- Having an abstract function `_getQuote()` that must be implemented by the derived adapters.
- Defining the `getQuote()` and `getQuotes()` as returning the output of the `_getQuote()` function.
- Defining a utility `_getDecimals()` function to fetch the decimals of a token.

2.2.3 SwaapSafeGuardOracle

The Swaap Adapter implemented as the `SwaapSafeGuardOracle` contract inherits the `BaseAdapter` contract and implements `_getQuote()` to calculate the price of Swaap SafeguardPool LP tokens.

First, the adapter calculates the total pool value by fetching the prices of the underlying tokens in the pool using the `EulerRouter`. Next, it determines the total supply of the pool through the private helper function `_getPoolSupplyAfterFees()`. This function increases the current supply of LP tokens with

the additional tokens that will be minted when the already accumulated fees are claimed. Finally, the adapter calculates the unit price of the LP token by dividing the total pool value by the adjusted total supply, which includes the unclaimed fees.

2.2.4 Roles & Trust Model

The `SwaapSafeGuardOracle` adapter is permissionless and deployed immutably. However, it relies on the security of the underlying SafeguardPool and the Euler Price Oracle system.

The `signer` role of SafeguardPool is untrusted. It should be possible for lenders to accept LP tokens of SafeguardPools as collateral without trusting the `signer`.

The tokens in the pool are assumed to have safe Euler oracles.

The `oracle` variable of `SwaapSafeGuardOracle` is assumed to be the `EulerRouter`.

Since (Version 2), `SwaapSafeGuardOracle` can underestimate the value of the `SafeguardPool` LP tokens. `SwaapSafeGuardOracle` can therefore be used to price LP tokens as collateral, provided that the borrower monitors the health of their position, but not to price borrowable tokens, as it could result in under-collateralized lending.

2.2.5 New in Version 2

In (Version 2), SwaapSafeGuardOracle is considerably redesigned to avoid an issue with potential manipulation of Swaap SafeguardPool LP manipulation by the `signer`. Instead of simply using the current balances of tokens in the pool to measure the value of the pool, the minimum between the value using the current balances and the reference benchmark balances (HODL balances) is used.

- value with current balances: `currentUnitPrice = (price0 * currentToken0Amount + price1 * currentToken1Amount)/totalSupply`
- value with HODL balances: `benchmarkUnitPrice = (price0 * hodlToken0Amount + price1 * hodlToken1Amount)/totalSupply`.

The value `currentUnitPrice` better represents the current actual value of LP tokens. However, it is manipulable by the `signer`, who can inflate it momentarily with a big unfavorable trade, therefore pumping LP tokens, and then deflate it back to the starting value by repeating slightly favorable trades. To prevent the oracle price from being inflatable, the minimum of `currentUnitPrice` and `benchmarkUnitPrice` is taken.

`benchmarkUnitPrice` represents the LP token value according to a daily checkpoint that syncs `currentUnitPrice` and `benchmarkUnitPrice`. In SafeguardPool, every time `PERFOMANCE_UPDATE_INTERVAL` elapses (configurable from 0.5 to 1.5 days), `updatePerformance()` can be called and the benchmark value is updated.

The update consists in computing the current value of the pool, and scaling the previous benchmark (composed of virtual amounts of `token0` and `token1`) such that the overall value of the new benchmark balances matches the overall value of the current token balances. The proportions of benchmark (HODL) `token0` and `token1` amounts are kept constant, and set at pool initialization time. When the pool is checkpointed with `updatePerformance()`, the values are simply updated so that their overall value (computed in token0) is equal to the current one, but not their individual values.

As the exchange rates of `token0` and `token1` change over time, the values of `benchmarkUnitPrice` and `currentUnitPrice` will diverge, as each are correlated to the price of `token0` and `token1` but in different proportions.

Taking the minimum of `benchmarkUnitPrice` and `currentUnitPrice` offers therefore manipulation protection, since the benchmark values are not manipulable, and inflating the balances will have no effect on the oracle price since the smaller benchmark values will apply. However, using `benchmarkUnitPrice` might result in an oracle price significantly lower than the *actual* LP values, and the oracle price will suddenly jump to the actual value when the benchmark is checkpointed.

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 the 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 defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. 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 that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those 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 severity. 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
- Correctness: Mismatches between specification and implementation

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	2

5.1 Non-Linear Pricing Handling in _getSafeguard

Correctness Low Version 2 Risk Accepted

CS-SWAAP-EPO-006

In the `_getSafeguardBenchmarkUnitPrice` function, the code calculates the unit price of the safeguard pool token by summing the quote values of the HODL balance per pool token (`hodlBalancePerPT`) for each underlying asset. It does this by calling the oracle with the unscaled `hodlBalancePerPT` amounts and then divides the oracle's output by a scaling factors (`hodlScale`) to adjust for decimal differences between the `hodlBalancePerPT` (18 decimals) and the underlying tokens (variable amount of decimals).

```
IPriceOracle(oracle).getQuote(hodlBalancePerPT0, address(tokens[0]), quote) / hodlScale0;
```

While this approach might be mathematically precise under the assumption of linear pricing — where the price per unit of token remains constant regardless of the input amount — it can lead to inaccuracies if the oracle's pricing is non-linear. In non-linear pricing models, the price per unit can vary depending on the input amount due to factors like liquidity depth, slippage, or tiered pricing structures.

Euler Price Oracles in theory support "size-aware pricing", where the quote output depends on `inAmount` non-linearly, as described in the Euler Price Oracle specification.

By calling the oracle with the unscaled `hodlBalancePerPT` and then dividing the result, the code assumes that scaling the output is equivalent to scaling the input. However, in a non-linear pricing oracle, this assumption doesn't hold true because the oracle may provide different price quotes for different input amounts.

Risk Accepted:

Swaap Labs notes that the deployed Euler oracles, which retrieve quotes for underlying tokens, operate on a linear pricing model. This means the current implementation aligns with the behavior of these oracles.

5.2 Potentially Significant Underpricing in Some Scenarios

Correctness | Low | Version 2 | Risk Accepted

CS-SWAAP-EPO-007

The `_getSafeguardBenchmarkUnitPrice` function in `SwaapSafeguardOracle` computes the price of the safeguard pool token based on the `HODL` balances. These balances are updated uniformly according to the overall performance of the pool as visible in the following snippet:

```
hodlBalancePerPT0 = hodlBalancePerPT0.mulDown(currentPerformance);
hodlBalancePerPT1 = hodlBalancePerPT1.mulDown(currentPerformance);
```

As a result, the computed token proportion using the `HODL` values remain static, matching the initial pool composition and do not account for token swaps or other changes over time. When the price of one token decreases significantly and that token is overrepresented by the `HODL` balances, the `_getSafeguardBenchmarkUnitPrice` may underestimate the pool's value compared to the `_getSafeguardCurrentUnitPrice`, which uses the actual current balances. Since the oracle returns the minimum of these two unit prices, the lower `HODL`-based price will be returned, leading to a price lower than expected. Because the code uses the minimum, this issue can only lead to underpricing, not overpricing.

Example Scenario:

- **Initial State**: The pool begins with 100 units of token0 and 100 units of token1, each initially valued at $1, for a TVL of $200.
- **Pool Activity:** Before the next update to the `HODL` balances (`_updatePerformance()` in `SafeguardPool`), all units of token0 are swapped for token1, resulting in a pool holding only token1 with a total value still at $200.
- **Price Change:** The price of token0 falls to $0. Calculation Outcomes:
  - `_getSafeguardBenchmarkUnitPrice` returns $100 because it still calculates a pool value based on the initial `HODL` ratios.
  - `_getSafeguardCurrentUnitPrice`, which uses actual pool balances would accurately reflect the TVL of $200.

Pool value is evaluated 50% below the it's actual value, to which it will jump as soon as the next checkpoint happens in the `SafeguardPool`. When the pool LP token is used as collateral, this could lead to unexpected liquidations for a state that is only transient.

Risk Accepted:

Swaap Labs notes that they will be the sole supplier of this collateral. They indicate they will apply risk management best practices to ensure their collateral remains within a safe range.

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	1

- Oracle Manipulation by the Signer Code Corrected

Medium-Severity Findings	1

- Intermediate Price Feed Decimals Incorrectly Specified Code Corrected

Low-Severity Findings	1

- Precision Loss Specification Changed

Informational Findings	1

- Inconsistent Use of PRICE_DECIMALS and lpTokenDecimals Code Corrected

6.1 Oracle Manipulation by the Signer

Design High Version 1 Code Corrected

CS-SWAAP-EPO-002

Euler Vaults using Swaap Safeguard Pool LP tokens as collateral should be aware that pricing LP tokens with `SwaapSafeguardOracle` requires complete trust in the `signer`. The off-chain quoting system inadvertently grants the `signer` the power to artificially inflate the value of the LP tokens in an unbounded way. This means the `signer` can manipulate the collateral value, and potentially empty the lender's vaults by borrowing assets against inflated LP tokens.

The problem arises from using the *spot* balances of underlying tokens to price the LP token, and is not inherent to Swaap SafeguardPool itself but it depends on how `SwaapSafeguardOracle` operates.

In a single transaction, a `signer` can borrow the maximum allowable amount of assets on Euler through the following steps:

1. **Inflating the Pool's Value**: The `signer` provides themselves with an unfavorable quote, effectively allowing them to donate a significant amount of `tokenIn` for a minimal amount of `tokenOut`. This inflates the pool's TVL which in turn increases the LP token value.
2. **Borrowing Against Inflated LP Tokens**: The `signer` then uses these overvalued LP tokens as collateral on Euler to borrow assets up to the maximum allowable amount.
3. **Restoring the Pool's Original State**: The `signer` provides quotes favorable to himself to reverse the initial donation, retrieving their initial `token0` and restoring the pool's reserves to their original state. Although the Fair Pricing Safeguard limits each swap to a 3% deviation from the on-chain price, the `signer` can make multiple small swaps to cumulatively reverse the initial donation. The other safeguards don't apply here as the HODL value as not been updated.

Example Scenario Starting with a pool containing 100 `token0` and 100 `token1`:

- **Initial State**:

- Pool Reserves: 100 `token0`, 100 `token1`, price is $1 for both tokens.
- LP Token Value: Based on the pool's TVL of $200.

- **Step 1 - Inflate Pool Value**:
  - Provides a self-serving quote to swap 100 `token0` for 0 `token1`, effectively donating 100 `token0` to the pool.
  - New Pool Reserves: 200 `token0`, 100 `token1`.
  - The pool's TVL increases to $300, increasing the LP token value by 50%.
- **Step 2 - Borrow Against Inflated Collateral**:
  - The `signer` supplies the overvalued LP tokens as collateral on Euler.
  - Borrows assets equivalent to the inflated LP token value.
- **Step 3 - Restore Pool Reserves**:
  - The `signer` provides favorable quotes to themselves to swap small amounts of `token1` back for `token0`.
  - Due to the 3% Fair Pricing Safeguard limit per swap, he performs multiple swaps to gradually extract the 100 `token0` initially donated.
    - Swap 1: Get 100 `token0` for 97 `token1`. Pool balance: 100 `token0`, 197 `token1`.
    - Swap 2: Get 97 `token1` for 94 `token0`. Pool balance: 194 `token0`, 100 `token1`.
    - Swap 3: Get 94 `token0` for 91 `token1`. Pool balance: 100 `token0`, 191 `token1`.
    - ...
    - Swap n: One token as a balance of 100 `token` and the other has `100 + (0.97)^n * 100`.
  - After enough swaps, the `signer` retrieves most of their initial 100 tokens.

This can be optimized to retrieve the entire initial donation **efficiently** by using batch swaps and abusing the 5% performance tolerance.

Code corrected:

The oracle has been fundamentally redesigned to return the minimum of the value of the pool at the current balances and the value of the pool at last checkpoint.

6.2 Intermediate Price Feed Decimals Incorrectly Specified

Correctness | Medium | Version 1 | Code Corrected

CS-SWAAP-EPO-001

`SwaapSafeguardOracle` uses the `calcScale()` and `calcOutAmount()` library functions of the Euler Price Oracles system to perform conversions between values with different decimals. The value `feedDecimals` should match the number of decimals of the `unitPrice` variable, which has `quoteDecimals`, however it is hardcoded as `PRICE_DECIMALS` which is 18. When `SwaapSafeguardOracle` is configured with a `quote` token which has a different decimals value than 18, this will result in incorrect price calculation.

`ScaleUtils.calcScale()` is used in the constructor of `SwaapSafeguardOracle` to compute a `scale`, which contain two packed values which represent the scaling factors (`10^decimals`) required to compute an output token amount (with `quoteDecimals`) from an input token amount (with `baseDecimals`) and a price feed (`feedDecimals`).

When using `calcScale()` the following parameters must be specified:

- `baseDecimals`, the decimals of the `base` token, the denomination of the amount that is appraised by the oracle, in this case the LP token of a Swaap Safeguard pool.
- `quoteDecimals`, the decimals of the `quote` token, the denomination to which the oracle converts the input `base` token. Configurable in the constructor.
- `feedDecimals`, the decimals of the intermediate price feed that is used to convert from `base` tokens to `quote` tokens.

The `scale` parameter computed with `calcScale()` is used in `calcOutAmount()`. When using `calcOutAmount` to convert from `base` tokens to `quote` tokens (`inverse == False`). We wish to perform the following mathematical operation:

$O_q = I_b \cdot P$

That is, O_q is the output amount of `quote` tokens, which is equal to I_b, the input amount of base tokens, times P, `unitPrice` or the exchange rate. Having to deal with fixed precisions and different decimals between the three values, this translates into the following code:

```
outAmount = inAmount * unitPrice * 10**quoteDecimals / (10**baseDecimals * 10**feedDecimals)
```

In practice `10**quoteDecimals` is called `priceScale` in `calcOutAmount()` internals, and `10**baseDecimals * 10**feedDecimals == 10**(baseDecimals + feedDecimals)` is called `feedScale`. The code is implemented in `ScaleUtils.sol:75` as:

```
outAmount = fullMullDiv(inAmount, priceScale * unitPrice, feedScale)
```

As described above, the `calcScale` parameter `feedDecimals` should match the decimals of `unitPrice`, the intermediate price obtained by dividing `poolTVL` by `totalSupply`.

```
uint256 unitPrice = poolTVL * 10 ** PRICE_DECIMALS / totalSupply;
```

since `poolTVL` is in `quote` tokens which has `quoteDecimals`, and `PRICE_DECIMALS` is equal to the decimals of `totalSupply`, the resulting `unitPrice` is in `quoteDecimals` precision. `feedDecimals` however has been specified in the constructor as `PRICE_DECIMALS` a constant of value `18`, instead of `quoteDecimals`, which is the correct value. This will result in catastrophic mispricing of tokens when the `quote` denomination has different amount of decimals than 18.

Code corrected:

Swaap Labs now correctly specifies `feedDecimals` in `calcScale()` as `quoteDecimals` instead of the incorrect `PRICE_DECIMALS`.

6.3 Precision Loss

Correctness **Low** **Version 1** **Specification Changed**

CS-SWAAP-EPO-003

When the oracle is configured with a low-precision quote token, e.g. USDC which has only 6 decimals, this limited precision is used to computed the unit price of the LP token, which potentially leads to a loss of precision in the final quoted amounts.

Indeed, the unit price for one (10**18 weis) LP token is computed in the precision of `quote` token as:

```
uint256 unitPrice = poolTVL * 10 ** PRICE_DECIMALS / totalSupply;
```

where `PRICE_DECIMALS == 18` and `totalSupply` has 18 decimals of precision. If the price of 1 LP token (10**18 weis) is in the same order of magnitude as 1 wei of the quote token (for example $1e-6 for USDC or $5e-4 for WBTC), then the LP price `unitPrice`, which is quoted in the low precision `quote` token, will round down with a significant relative effect.

Example:

- `quote` token USDC
- `totalSupply`: `1_000_000_000e18`, 1 billion LP tokens minted
- `poolTVL` : `999_999_999`, slightly less than $1000

The resulting `unitPrice` will be computed as `999_999_999 * 10**18 / (1_000_000_000 * 10**18) == 0`. Then even if the price of 500M LP tokens is requested, since the unit price rounds down, `0` will be returned.

Specification changed:

Swaap Labs acknowledges the issue, and addresses it by documenting the behavior and warning future deployers to use a `quote` token with high decimal precision.

6.4 Inconsistent Use of PRICE_DECIMALS and lpTokenDecimals

Informational **Version 1** **Code Corrected**

CS-SWAAP-EPO-004

In the `SwaapSafeguardOracle` contract, the constant `PRICE_DECIMALS` is set to 18 and is used in the calculation of `unitPrice` at line 75:

```
uint256 unitPrice = poolTVL * 10 ** PRICE_DECIMALS / totalSupply;
```

The purpose of `PRICE_DECIMALS` here is to adjust the precision to ensure that the final value of `unitPrice` matches the precision of the `poolTVL` variable. This effectively "cancels out" the decimals of `totalSupply`.

However, `totalSupply` is an amount of LP tokens, and the decimals of LP tokens are already retrieved in the constructor through the `_getDecimals()` function. In SafeguardPool they are guaranteed to be 18 decimals. It is not clear what is the purpose of `PRICE_DECIMALS`, given that an invariant holds according to which `lpTokenDecimals` should have the same value as `PRICE_DECIMALS`.

Code corrected:

Swaap Labs renamed constant `PRICE_DECIMALS` to `LP_DECIMALS`, and asserts that the LP token indeed has `LP_DECIMALS` in the constructor.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Incorrect Comment

Informational **Version 2**

CS-SWAAP-EPO-008

In the `_getSafeguardCurrentUnitPrice` function, at line 131, the comment incorrectly states that the TVL is calculated by summing the **USD value** of each pool token. However, the code actually sums the values in terms of the **quote asset**, which may not be USD.

7.2 Reentrancy Vulnerabilities in Balancer V2

Informational **Version 1**

CS-SWAAP-EPO-005

The Balancer V2 infrastructure, on which Swaap's SafeguardPool is built, has a known read-only reentrancy vulnerability.

SafeguardPool addresses this vulnerability directly, allowing the SafeguardPool Oracle to omit the typical reentrancy guard, the Balancer V2 Vault context check. However, this approach relies on the assumption that all potential reentrancy entry points within Balancer V2 have been identified and mitigated.

If additional reentrancy vulnerabilities are discovered in Balancer V2 the future, the SafeguardPool Oracle could be exposed.