PUBLIC

Code Assessment

of the KyberSwap Elastic V2 Smart Contracts

May 16, 2023

Produced for

Kyber Network
On-chain Liquidity Protocol

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Kyber Team,

Thank you for trusting us to help Kyber Network with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of KyberSwap Elastic V2 according to Scope to support you in forming an opinion on their security risks.

Kyber Network implements an AMM that allows liquidity providers to concentrate the liquidity in a certain price range, with the fees being automatically reinvested in the second constant product curve without concentrated liquidity. On top of the AMM, Kyber Network implements the anti-sniping mechanism to mitigate the issue of just-in-time liquidity provision, and a TWAP oracle for each pool.

The most critical subjects covered in our audit are functional correctness, access control, and precision of arithmetic operations. Security regarding all the aforementioned subjects is good.

The general subjects covered are code complexity, trustworthiness, gas efficiency and documentation. Security regarding all the aforementioned subjects is high.

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	0
Low-Severity Findings	6
• Code Corrected	5
• Risk Accepted	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 KyberSwap Elastic V2 repository based on the documentation files. This assessment is performed on the modified codebase from KyberSwap Elastic report. The scope of this review is focused on the changes in the files from Scope, compared to the last commits of the KyberSwap Elastic report. A focus was done on the `Pool` contract.

For this audit, the following files in the `contracts` folder are in scope:

- All files in `interfaces` subfolder, if not mentioned in Excluded from Scope.
- All files in `libraries` subfolder, if not mentioned in Excluded from Scope.
- All files in `periphery` subfolder, if not mentioned in Excluded from Scope.
- oracle/PoolOracle.sol
- Factory.sol
- Pool.sol
- PoolStorage.sol
- PoolTicksState.sol

Open issues and Notes reported in the report of KyberSwap Elastic are not repeated in this report but may still apply. Please refer to report of the KyberSwap Elastic review.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	24 April 2023	1902450fd9bcbc39c8cf53b0570837513d32cdfb	Initial Version
2	10 May 2023	5c5f87619544e29df0af35dfb5fb98176c18b22b	Updated Version
3	15 May 2023	3ba84353cbd88f30f222bb9c673e242a2e46fd12	Version with fixes

For the solidity smart contracts, the compiler version `0.8.9` was chosen.

2.1.1 Excluded from scope

Every contract not explicitly listed above and third party libraries are out-of-scope. Especially:

```
* interfaces/periphery/IQuoterV2.sol
* interfaces/IWETH.sol
* libraries/FullMath.sol
* libraries/TickMath.sol
* All files in ``mock`` subfolder of ``contracts`` folder.
* All files in ``echidna`` subfolder of ``contracts`` folder.
* periphery/libraries/BytesLib.sol
* periphery/libraries/PoolTicksCounter.sol
* periphery/QuoterV2.sol
```

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.

KyberSwap Elastic V2 is a version of noncustodial dynamic market maker protocol implementation, that is similar to Kyber DMM v1 and other AMM protocols. It differs from Kyber DMM v1 in two main ways:

1. *Concentrated liquidity*: similar to Uniswap V3 protocol, KyberSwap Elastic V2 allows liquidity providers (LPs) to provide liquidity into a specific price range. This allows more effective liquidity utilization for the LPs.
2. *Reinvestment curve*: this curve allows LP fees to be automatically reinvested into the pool, thus achieving the compounding interest for LP position.

The main contracts of the KyberSwap Elastic V2 are:

- Factory
- Pool
- Router
- AntiSnipAttackPositionManager
- PoolOracle

2.2.1 Factory

Factory provides governance fee destination and percentage via `feeConfiguration` function. Factory contract creates new Pool contracts for given pair of tokens and swap fee. The implementation code of new Pool contracts that the factory creates cannot be updated. Pool contracts themselves are also not upgradeable. Factory also stores all whitelisted position managers for the Pool contracts. Factory has one privilege role: `configMaster`. Holder of this role can:

- Change configuration master
- Enable or disable position manager whitelisting
- Adding new position manager contracts to the whitelist
- Update Vesting period duration (Used by `AntiSnipAttackPositionManager`)
- Change governance fee and governance fee recipient. The governance fee cannot be higher than 20%.
- Adding new fee values and distances that pools can support.

2.2.2 Pool

The `Pool` contract implements the AMM with concentrated liquidity. For liquidity provision, it allows whitelisted addresses, typically the `AntiSnipAttackPositionManager` to mint new positions or modify existing ones on the pool with `mint()`. The `burn` function is permissionless and allows the owner of the position to partially or fully de-provision their position. Interaction with `mint()` and `burn()` will send the owed amount of reinvestment tokens to the `msg.sender`. Holders of `RTokens` can then redeem them with `burnRTokens()` for the underlying tokens of the pool.

Regular users can use the `swap` function to swap between the pool's underlying assets, the fees collected during a swap are reinvested in the reinvestment curve and `RTokens` are minted for the liquidity providers and the governance. For a flash loan, users can use the `flash` function to borrow part of or all the assets from the pool. The fees collected on a flash loan are sent to the governance and are not reinvested, and thus do not contribute to the LP fees growth.

2.2.2.1 Formulas

For the current amount of T_0 and T_1, a `Pool` implements a constant product automated market maker with formula

$$x * y = (L_b + L_r)^2$$

, where L_b is an aggregated liquidity from all DMM LPs positions that provide liquidity for the current price p_c and L_r is liquidity provided by the reinvestment curve.

Concentrated liquidity provision is possible at specific price ranges. Each price is linked to a tick. The price at a tick `t` is given by

$$\sqrt{1.0001^t}$$

See Ticks section for more info about ticks. When an initialized tick is crossed, the active base liquidity L_b is updated to represent the new aggregated liquidity available at the new price.

All fees collected from swaps effectively increase the L_r amount. Part of this fee goes to the governance address. The government fee percentage and receiver configuration are stored on the Factory contract. The maximum government fee is 20% of the swap fees. When a swap crosses a tick or when users add/remove liquidity from the pool, the reinvestment tokens (`RTokens`) are minted for the DMM position owners. The minted `RTokens` are ERC20 tokens that can be transferred and burned to get a share of reinvestment curve liquidity.

The `Pool` contract supports flash loan functionality. The flash loan fee is the same as the swap fee and the full fee amount is sent to the Factory defined governance fee destination address.

The formulas for price computations are (1.) for the current price p_c, and (2.) for the target price p_t:

$$1.\ p_c = \frac{y}{x}$$

$$2.\ p_t = \frac{y + \Delta y}{x + \Delta x}$$

The formulas for a swap are (3.) for T_0 to T_1, and (4.) for T_1 to T_0:

$$3.\ (x + (1 - fee) * \Delta x) * (y + \Delta y) = (L_b + L_r)^2$$

$$4.\ (x + \Delta x) * (y + (1 - fee) * \Delta y) = (L_b + L_r)^2$$

Each swap will reinvest the collected fees in the reinvesment curve, so the invariant is updated at each swap step. The formula to compute the new invariant after a swap is:

$$(x + \Delta x) * (y + \Delta y) = (L_b + L_r^{'})^2$$

The formulas to compute the new liquidity of the reinvestment curve after the swap are (5.) for T_0 to T_1, and (6.) for T_1 to T_0:

$$5.\ \sqrt{(x_r + fee * \Delta x) * y_r} = L_r^{'\Delta x}$$

$$6.\ \sqrt{x_r * (y_r + fee * \Delta y)} = L_r^{'\Delta y}$$

Since the square root is costly to compute on a smart contract, Kyber Network implements approximations for (5.) and (6.) that are resp. (7.) and (8.):

$$7.\ L_{r_{approx}}^{'\Delta x} = L_r + \frac{fee * \Delta x * \sqrt{p_c}}{2}$$

$$8.\ L_{r_{approx}}^{'\Delta y} = L_r + \frac{fee * \Delta y}{2 * \sqrt{p_c}}$$

The amount of `RTokens` that are minted represents the active DMM position's participation in the increase of the reinvestment curve's liquidity:

$$calcrMintQty = \frac{L_b^{'}}{L_b^{'} + L_{r_{approx}}^{'}} * \frac{L_{r_{approx}}^{'} - L_r}{L_r} * TotalSupply_{RTokens}$$

2.2.3 Ticks

For a `Pool` with a `tickDistance` equal to t_d, an initialized tick `t` will be responsible for the prices in `[a, b)` with:

$$a = \sqrt{1.0001^t}, b = \sqrt{1.0001^{t+t_d}}$$

Each initialized tick is updated when they are entered from below, left from below, or modified due to an LP position tweak. Each initialized tick holds information about the LP positions using that tick as a boundary (up or down):

- `liquidityGross`: positive value. The sum of the liquidity of all the positions having this tick as a boundary (up or down).
- `liquidityNet`: can be positive or negative. Active liquidity delta to be added/removed to/from the active liquidity L_b when the tick is crossed. If the tick `t` is crossed up, the net liquidity from tick `t+1` is added, if the tick `t` is crossed down, the net liquidity from tick `t` is deducted. The value added by an LP is negative/positive when the tick is an upper/lower tick.
- `feeGrowthOutside`: yields a value such that the difference between a range's upper and lower ticks' `feeGrowthOutside` is equal to the fee growth inside the range
- `secondsPerLiquidityOutside`: yields a value such that the difference between a range's upper and lower ticks' `secondsPerLiquidityOutside` is equal to the seconds elapsed per unit of active base liquidity inside the range

To help the computation of `feeGrowthOutside` and `secondsPerLiquidityOutside`, the pool tracks the two values `feeGrowthGlobal` and `secondsPerLiquidityGlobal`, holding the global growth of the fee and the seconds elapsed per active unit of base liquidity L_b over the whole pool.

2.2.4 Router

The `Pool` contracts rely on callbacks to get the funds from the message sender. The `Router` contract acts as a service contract, that allows using token approvals to fulfill the callback request from pool. In addition, using the swap path data, the user can perform a chain of swaps between multiple pairs of tokens.

2.2.5 AntiSnipAttackPositionManager

A snipping attack is an attack vector for concentrated liquidity pools. It is also known as : Just-in-Time Liquidity (JIT). A liquidity provider can add and remove liquidity atomically in one block, sandwiching the swap transactions. This way, the LP gains the majority of the swap fees, while not having a long-term commitment to liquidity provision. `AntiSnipAttackPositionManager` is a contract that prevents this, by introducing a vesting period for the acquired fees. The contract will distribute a unique ERC721 token for every position LPs open. `AntiSnipAttackPositionManager` contract will act as a direct liquidity provider for the pool and will receive and hold the `RTokens` from fees. It does so by locking aside the appropriate part of `RTokens` and paying out the vested `RTokens`. The amount of withdrawable fees linearly grows during the vesting period, which is defined in the `Factory` contract. If the position is removed before the end of the vesting period, tokens that are still locked will be burned without profit. Effectively, this prevents the creation and destruction of the liquidity position in the same block and does not allow the malicious LPs to avoid the impermanent loss risk.

2.2.6 PoolOracle

The `PoolOracle` contract implements a Time Weighted Average Price (TWAP) oracle. The oracle works in the same way Uniswap V3 TWAP oracle works and can be used to indicate the approximated geometric average price of a pair of assets on a given pool. The oracle can track the price of multiple pairs simultaneously by tracking a mapping indexed by `msg.sender`. It yields a finite number of observations (`cardinality`) per pool, at most one observation can be recorded per block, and the latest observation has the cumulative tick value of

$$\sum_{i=1}^{n} observationTick_i * observationTime_i - observationTime_{i-1}$$

with `observationTick`$_0$ `= 0` and `observationTime`$_0$ `= block.timestamp` at oracle initialization.

A new observation is triggered from the `Pool` when:

- a tick (initialized or not) is crossed during a swap, the oracle is updated with the tick before the swap
- an LP position is updated (`_tweakPosition()` is called)

To have access to historic prices, one can voluntarily pay for the initialization of more observation slots by calling the function `increaseObservationCardinalityNext`.

The following functions are available to query the cumulative tick values:

- `observeFromPool`: get the value of the accumulator at different points in time starting from `now` (`[now-secondsAgos`$_0$`, now-secondsAgos`$_1$`, ...]`) in a given pool
- `observe`: get the value of the accumulator at different points in time starting from a given `time` (`[time-secondsAgos`$_0$`, time-secondsAgos`$_1$`, ...]`) in the pool that has the address `msg.sender`
- `observeSingle`: get the value of the accumulator at one point in time starting from a given `time` (`[time-secondsAgo`) in the pool that has the address `msg.sender`
- `observeFromPoolAt`: get the value of the accumulator at different points in time starting from a given `time` (`[time-secondsAgos`$_0$`, time-secondsAgos`$_1$`, ...]`) in in a given pool

2.2.7 Trust model

- `configMaster`: is trusted to act non-maliciously and to the advantage of the system and the users by setting reasonable parameters and whitelisting trusted addresses
- Pool deployed and unlocker: trusted
- liquidity providers: not trusted
- users: not trusted

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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	1

- DOMAIN_SEPARATOR Is Not Recomputed if chainId Changes Risk Accepted

5.1 DOMAIN_SEPARATOR Is Not Recomputed if chainId Changes

Security Low Version 1 Risk Accepted

CS-KYBE2-003

The `ERC712Permit.DOMAIN_SEPARATOR` is immutable, and thus won't be changed if the chain forks. If Ethereum fork in the future (like PoW fork), the chainId will change however the BasePositionManager on forked chain will still accept permit with old chainId. This leads to cross-chain replay attacks, where signature from one domain is used on the other domain.

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 Observation Functions Parameters Code Corrected

Medium-Severity Findings	0

Low-Severity Findings	5

- Compiler and Library Versions Code Corrected
- Missing Sanity Checks Code Corrected
- Swap Amount Vs Price Limit Discrepancy Code Corrected
- maxNumTicks Computation Can Be Wrong Code Corrected
- secondsPerLiquidity of the First LP Starts at UNIX Time 0 Code Corrected

6.1 Oracle Observation Functions Parameters

Correctness High Version 1 Code Corrected

CS-KYBE2-001

The `PoolOracle` functions `observe`, `observeSingle`, and `observeFromPoolAt` accept arbitrary parameters `time` that should serve as a reference point for the `secondsAgo` parameter, and `tick` that should be used to transform the latest observation if needed. But the `Oracle` library requires the provided `time` to be the current block timestamp, and `tick` to be the current tick of the pool. More specifically for `time`, the function `Oracle.lte` requires `a` and `b` to be chronologically before `time`. Thus, an arbitrary `time` parameter may return a wrong value for the accumulator. The same is valid for an arbitrary value of `tick`, which could yield an incorrect accumulator if the last observation had to be transformed.

Example with arbitrary `time`:

```
cardinality = 8
block.timestamp = 1050
time = 550
secondsAgo = 100
```

With the following state, for simplicity assume that $\texttt{tick}_i$ ==$\texttt{observationTimestamp}_i$, only the timestamps are showed:

```
|350| |500| |700| |900| |1024| |150| |220| |300|

                                ^index
```

the function `observeSingle(550, 100, 1024)` will yield surrounding observations `(4,0)` (index `4` for `beforeOrAt` and index `0` for `atOrAfter`), instead of the expected `(0,1)`, and return a wrong `tickCumulative` value.

Description of changes:

Remove `observeFromPoolAt`, `observe`, and `observeSingle` functions, add `observeSingleFromPool` to read a single observation from a pool. All observe functions use `block.timestamp` as a `time` for.

6.2 Compiler and Library Versions

Design | Low | Version 1 | Code Corrected

CS-KYBE2-002

Solc version `0.8.9` is not the most up-to-date version and has known bugs.

The smart contract libraries used by the project are:

```
"@openzeppelin/contracts": "4.3.1",
"@openzeppelin/contracts-upgradeable": "^4.6.0",
```

However, these libraries are neither up to date nor consistent with one another.

Code corrected:

The OZ libraries now both use version `4.3.1`.

Regarding the solc compiler Kyber Network responded:

> We didn't upgrade the solidity version to latest as it could increase the possible changes for the protocol.

Known bugs in solc `0.8.9` should not be triggered the assessed codebase.

6.3 Missing Sanity Checks

Design | Low | Version 1 | Code Corrected

CS-KYBE2-004

The function `TicksFeesReader.getNearestInitializedTicks` is missing input sanitization for the `tick` parameter. It can accept invalid ticks such that `tick < MIN_TICK` or `tick > MAX_TICK`. The while loops won't terminate for invalid ticks.

Code corrected:

A check was added.

```
require(T.MIN_TICK <= tick && tick <= T.MAX_TICK, 'tick not in range');
```

6.4 Swap Amount Vs Price Limit Discrepancy

Correctness | Low | Version 1 | Code Corrected

CS-KYBE2-005

The swap terminates in 2 cases: specified amount is exhausted or specified price limit is reached. However, there exists an edge case when specified amount is just enough to reach a price limit. In that case the Pool will rely on specified amount value as a limit, that will lead to computation of a new pool state using `estimateIncrementalLiquidity` function. If the price limit was used, the new state computation would be handled by `calcIncrementalLiquidity` function. The pool state is defined by prices and computation of a new state using token amounts leads to more numeric conversions and thus to less precision.

If a Pool has following initialized tick ranges: [a, b) [b, c). And current tick is b+1, a swap specifying `getSqrtRatioAtTick(b)` as a limit would switch the liquidity to the value of [a, b) tick range. But a swap `swapQty` needed to reach the same state would result in a pool state where the liquidity has not being shifted.

Code corrected:

The `computeSwapStep` function uses `calcIncrementalLiquidity` when the `usedAmount` is equal to `specifiedAmount`. Thus, the more precise price limit is used for this edge case.

6.5 `maxNumTicks` Computation Can Be Wrong

Design | Low | Version 1 | Code Corrected

CS-KYBE2-006

In the functions `TicksFeesReader.getTicksInRange` , the computation of `maxNumTicks` can return a value that is too low when `length==0`, thus making the returned memory array incomplete.

Example, when `startTick < 0`:

```
MAX_TICK = 2;
MIN_TICK = -2;
length = 0;
startTick = -1;
tickDistance = 1;
```

With this setting, `maxNumTicks=3` and only the ticks `-1, 0, 1` will be returned, missing the tick 2. In `getAllTicks` for this case will be: `maxNumTicks=7`, while should be 5.

Example, when `startTick > 0`:

```
MAX_TICK = 5;
MIN_TICK = -5;
length = 0;
startTick = 2;
tickDistance = 2;
```

With this setting, `maxNumTicks=1` and only the tick 2 will be returned, missing the ticks 4 and 5.

Code corrected:

The cases from above are fixed.

6.6 `secondsPerLiquidity` of the First LP Starts at UNIX Time 0

Correctness Low Version 1 Code Corrected

CS-KYBE2-007

When a liquidity provider (LP) opens the first position (LP_1) of a pool at t_1, `poolData.secondsPerLiquidityUpdateTime == 0` and `_syncSecondsPerLiquidity()` will have no effect since no base liquidity is yet in the pool. When the second position is opened at t_2, `_syncSecondsPerLiquidity()` will update the state, but `secondsElapsed` will be equal to the time delta from UNIX timestamp `0` until now (t_2). So, the liquidity added by LP_1 will be accounted for since `0` instead of t_1.

Description of changes:

Always update the `poolData.secondsPerLiquidityUpdateTime` to the current block timestamp whenever the secondsElapsed > 0.

6.7 Code Duplication

Informational Version 1 Code Corrected

CS-KYBE2-008

In the case `!isToken0`, the function `SwapMath.calFinalPrice` computes the same `tmp` value in each of the subbranches. The computation can be carried out outside of the conditional structure.

Code corrected:

The common code was moved outside the branch bodies.

6.8 Wrong Comments

Informational Version 1 Code Corrected

CS-KYBE2-012

The `natspec` of the struct `IBasePositionManager.MintParams` still mentions the `fee` in `bps`, but the fees have been updated to be in `feeUnits`.

Code corrected:

`@param fee` now correctly states that fee is in fee units.

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 Gas Griefing Attack

Informational **Version 1** **Risk Accepted**

CS-KYBE2-009

The swap function can perform multiple iterations of the while loop before terminating. Such execution can cost a lot of gas. Malicious actor can bring the pool price to an extremely high or low value. This can be done during the initial Pool unlock or via swap. While swap will require a lot of gas from attacker, similar amount of gas will also be required to bring the price back to true value. Since the amount of tokens needed to `unlockPool` is low, the cost of attack is small.

7.2 Oracle Limitations

Informational **Version 1** **Risk Accepted**

CS-KYBE2-010

The `tickCumulative` from PoolOracle contract can be used to compute the time-weighted average tick for a given period of time. If the price is computed from this tick, this is effectively a geometric mean of the time-weighted average price (gm-TWAP). Compared to the arithmetic mean TWAP (am-TWAP), gm-TWAP is more sensitive to upward price movements and less sensitive to downward price movements. Any protocol that plans to use PoolOracle needs to be aware of this.

In addition, in PoS consensus, the multi-block price manipulations are possible on AMM protocols:

- https://chainsecurity.com/oracle-manipulation-after-merge/
- https://blog.uniswap.org/uniswap-v3-oracles

7.3 PoolOracle Observations Mapping Collision

Informational **Version 1** **Risk Accepted**

CS-KYBE2-011

The `mapping(address => Oracle.Observation[65535])` field in PoolOracle contract allows any msg.sender to modifier consecutive 2**16 storage slots. This theoretically can write to storage slot 151 and thus overwrite the owner of the contract. Note that solidity does not check for storage pointer overflows. However, this is a practically impossible attack, since it requires attacker to find an address that corresponds to mapping storage slot with 240 fix bits.