PUBLIC

Code Assessment

of the KyberSwap Elastic Smart Contracts

December 06, 2021

Produced for

Kyber Network
On-chain Liquidity Protocol

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Loi, Dear Victor,

Thank you for trusting us to help Kyber Network with this security audit. Our executive summary is providing a holistic overview of KyberSwap Elastic to support you in forming an opinion on its smart contract security risk.

KyberSwap Elastic is an automated market maker (AMM) implementation, that allows liquidity providers to concentrate the liquidity in a certain price range.

The most critical audit subjects are functional correctness, external dependency integration and protection against adversarial agents. We found some deviations from the functional correctness which were reported. Regarding external dependency integration, we found minor mismatch from standard. Lastly, bugs that limited the AntiSniping (aka JIT liquidity provision) protection were reported.

The general audit subjects covered include trustworthiness, documentation, and gas efficiency. Regarding trustworthiness, while pools are not upgradable, there are certain system parameters like whitelisted position managers that can be set only by privileged ConfigMaster role holder. We found certain parts of the documentation that could be improved so that other projects can better integrate with the Kyber Network protocol. Lastly, minor possible improvements to gas efficiency were reported.

In summary, we find that the codebase at last version commit in Scope provides provides a high level of security. It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. Since the protocol logic is quite sophisticated, techniques such as property based testing and formal verification can bring valuable additional assurance. 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	3
• Code Corrected	3
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	10

• Code Corrected	7
• Specification Changed	3

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 following source code files inside the KyberSwap Elastic repository based on the documentation files:

- contracts/Factory.sol
- contracts/interfaces/callback/IFlashCallback.sol
- contracts/interfaces/callback/IMintCallback.sol
- contracts/interfaces/callback/ISwapCallback.sol
- contracts/interfaces/IFactory.sol
- contracts/interfaces/IPool.sol
- contracts/interfaces/periphery/IBasePositionManager.sol
- contracts/interfaces/periphery/IERC721Permit.sol
- contracts/interfaces/periphery/INonfungibleTokenPositionDescriptor.sol
- contracts/interfaces/periphery/IRouter.sol
- contracts/interfaces/periphery/IRouterTokenHelper.sol
- contracts/interfaces/periphery/IRouterTokenHelperWithFee.sol
- contracts/interfaces/pool/IPoolActions.sol
- contracts/interfaces/pool/IPoolEvents.sol
- contracts/interfaces/pool/IPoolStorage.sol
- contracts/libraries/BaseSplitCodeFactory.sol
- contracts/libraries/CodeDeployer.sol
- contracts/libraries/Linkedlist.sol
- contracts/libraries/LiqDeltaMath.sol
- contracts/libraries/MathConstants.sol
- contracts/libraries/QtyDeltaMath.sol
- contracts/libraries/QuadMath.sol
- contracts/libraries/ReinvestmentMath.sol
- contracts/libraries/SafeCast.sol
- contracts/libraries/SwapMath.sol
- contracts/periphery/AntiSnipAttackPositionManager.sol
- contracts/periphery/base/DeadlineValidation.sol
- contracts/periphery/base/ERC721Permit.sol
- contracts/periphery/base/ImmutablePeripheryStorage.sol
- contracts/periphery/base/LiquidityHelper.sol

- contracts/periphery/BasePositionManager.sol
- contracts/periphery/base/RouterTokenHelper.sol
- contracts/periphery/base/RouterTokenHelperWithFee.sol
- contracts/periphery/libraries/AntiSnipAttack.sol
- contracts/periphery/libraries/LiquidityMath.sol
- contracts/periphery/libraries/PathHelper.sol
- contracts/periphery/libraries/PoolAddress.sol
- contracts/periphery/libraries/TokenHelper.sol
- contracts/periphery/Router.sol
- contracts/Pool.sol
- contracts/PoolStorage.sol
- contracts/PoolTicksState.sol

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

V	Date	Commit Hash	Note
1	16 October 2021	f552ef1a2d89f16f7b7f195d56d1ced1eb151695	Initial Version
2	25 November 2021	520a775874f8e30feabcd44eebfdffc2ac35b928	Version with fixes
3	04 December 2021	650064b8d1a49083f6054111b82d994f51abec45	Second Version with fixes

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

2.1.1 Excluded from scope

All smart contracts that are not mentioned in the above section. The imported libraries and smart contracts that are not mentioned in the Scope sections were assumed to be adhering to their specification

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 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 allows liquidity provides (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 constant product formula for a fixed price `p = y/x` is represented as:

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

, where L_p is an aggregated liquidity from all DMM LPs positions that provide liquidity for a given price p and L_r is a liquidity provided by the reinvestment curve. All fees collected from swaps effectively increase the L_r amount. When a swap changes the L_p value (due to a p price change) or when users add/remove liquidity from the pool, the reinvestment tokens (RTokens) are minted for the DMM position owners. These RTokens can be burned in the Pool for a share of underlying collateral.

The main contracts of the KyberSwap Elastic are:

- Factory
- Pool
- Router
- AntiSnipAttackPositionManager

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 factory creates cannot be updates. Pool contracts themselves are also not upgradable. Factory also stores all whitelisted position managers for the Pool contracts. Factory has one privilege role: Configuration master. Holder of this role can:

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

2.2.2 Pool

LPs can deposit the funds into the Pool contract and provide the liquidity for swaps in a defined price range. Swaps effectively "takes" funds from msg.sender and "gives" them to the argument provided recipient. The "taken" amount also contains the fee amount that is deposited into the Pool as reinvestment curve liquidity. Part of this fee goes to the governance address. The minted RTokens are ERC20 tokens that can be transferred and burned to get the share of reinvestment curve liquidity.

Pool supports flash loan functionality. The flash loan fee % is same as swap fee and this fee goes to the factory defined governance fee destination address.

2.2.3 Router

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

2.2.4 AntiSnipAttackPositionManager

Snipping attack is a novel attack vector for concentrated liquidity pools. It is also known as : Just-in-Time Liquidity (JIT). A malicious liquidity provider can add and remove liquidity atomically in one block, sandwiching the swap transactions. This way the attacker gains majority of the swap fees, while having no impermanent loss risk. AntiSnipAttackPositionManager is a contract that prevents snipping attack, by introducing a vesting period for the acquired fees. The LPs who want to add liquidity, create a ERC721 unique token. AntiSnipAttackPositionManager contract will act as a direct liquidity provider on 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, that is defined on factory contract. If the position is liquidated before the end of vesting period, still locked tokens will be burned without profit. Effectively, this prevents creation and destruction of the liquidity position in the same block and does not allow the malicious LPs to avoid the impermanent loss risk.

2.2.5 Assumptions

During the assessment we relied on following assumptions:

- Factory ConfigMaster role holder is trusted non-malicious actor
- System parameters are coherent with one another and with global network parameters. For example, `vestingPeriod` parameter should be large enough to prevent a minimal risk snipping attack.

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:

- 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	0

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	3

- Bypassing Antisnipping Protection Code Corrected
- Function Pool.burnRTokens Return Values Code Corrected
- Locked Funds Remain Locked After vestingPeriod Update Code Corrected

Medium-Severity Findings	2

- Broken/Partial ERC165 Support Code Corrected
- Function Pool.unlockPool Reentrancy Code Corrected

Low-Severity Findings	10

- Function ERC721Permit.permit Payable Code Corrected
- Function Pool.burnRTokens Natspec Specification Changed
- Function Pool.burnRTokens Potential Reentrancy Code Corrected
- Function SwapMath.calcFinalPrice Rounding Down Code Corrected
- Gas Inefficiency in insert() Code Corrected
- Pool swap Max Tick Distance Code Corrected
- Position Manager Storage Access Code Corrected
- Solidity Compiler Pragma Code Corrected
- Specification Mismatches in SwapMath Specification Changed
- flash() Sends Fees to feeTo Specification Changed

6.1 Bypassing Antisnipping Protection

Security High Version 1 Code Corrected

The AntisnippingManager implements logic to protect against so-called liquidity-snipping (Just-in-Time Liquidity) attacks to prevent attackers from adding much liquidity before a swap and removing it right afterwards to collect most of the fees while not being exposed to LP risks.

Kyber Network removes the economic incentive of such an attack by locking fees for `vestingPeriod` which means an immediate withdrawal of liquidity should set the collected fees to zero.

Note, that AntiSnipAttack protection only comes in play if `feeGrowthInsideLast` of the position manager and the `feeGrowthInsideLast` of the position are not equal:

```
if (feeGrowthInsideLast != pos.feeGrowthInsideLast) {
    ....
    (additionalRTokenOwed, feesBurnable) = AntiSnipAttack.update(
    ...
}
```

Also, `feesBurnable` can only be non-zero if liquidity is removed:

```
if (isAddLiquidity) {
    ....
} else if (_self.feesLocked > 0) {
    feesBurnable = (_self.feesLocked * liquidityDelta) / uint256(currentLiquidity);
    _self.feesLocked -= feesBurnable;
}
```

Thus, the following attack is possible:

1. Attacker sees a huge swap and mints an enormous position
2. Swap occurs.
3. An attacker adds a small amount of liquidity. The position's `feeGrowthInsideLast` is updated. However, rTokens are now locked.
4. An attacker removes all his liquidity which does not enter the AntiSnipAttack code since there was no fee growth. Liquidity is withdrawn and rTokens remain locked.
5. After `vestingPeriod` has passed the attacker can withdraw the newly generated fees.

Even though the attacker does not immediately withdraw the fees, his liquidity came and went immediately while generating a temporarily locked profit for the attacker.

Code corrected:

In version 3 of the code, the Antisnipping protection logic is triggered on every call of `removeLiquidity` function.

6.2 Function `Pool.burnRTokens` Return Values

Correctness High Version 1 Code Corrected

Function `burnRTokens` of Pool contract has following definition:

```
/// @return qty0 token0 quantity sent to the caller for burnt reinvestment tokens
/// @return qty1 token1 quantity sent to the caller for burnt reinvestment tokens
function burnRTokens(uint256 qty, bool isLogicalBurn)
  external
  returns (uint256 qty0, uint256 qty1);
```

However the `qty0` and `qty1` value are not assigned in the implementation of this function. Thus 0 values will be returned instead.

The position managers rely on these return values as they implement slippage protection as follows:

```
(amount0, amount1) = pool.burnRTokens(rTokenQty, false);
require(amount0 >= params.amount0Min && amount1 >= params.amount1Min, 'Low return amounts');
```

Ultimately, the transaction will revert if `amount0Min > 0 && amount1Min >0` holds.

Code corrected:

The values are now properly assigned to the return variables.

6.3 Locked Funds Remain Locked After `vestingPeriod` Update

Correctness High Version 1 Code Corrected

The AntiSnipAttackPositionManager prevents profitable snipping attacks by locking rewards for a certain amount of time. The fees locked in a position with `tokenId` are stored in `AntiSnipAttackPositionManager:antiSnipAttack[tokenId].feesLocked`. However, if `vestingPeriod` is set to zero, `feesLocked` remains locked.

Assume the following scenario:

1. `vestingPeriod = 1 day`
2. User mints a position.
3. After 12 hours, the User adds liquidity to a position. Assume that 1 rToken in fees has been earned totally while half of it is locked.
4. `vestingPeriod` set to 0.
5. Whenever the user performs a position-modifying action, the following code gets executed.

   ```
   if (vestingPeriod == 0) return (feesSinceLastAction, 0);
   ```

6. Only the newly accumulated fees become claimable while the locked fees remain locked. Hence, 0.5 rTokens will be unclaimable.

Thus, changes to the vestingPeriod can potentially allow users withdrawal of more fees, than it was intended. Current AntiSnipAttackPositionManager and AntiSnipAttack library rely on constant vestingPeriod. To conclude, the AntiSnipAttack library should be aware that the vesting period for fees could change.

Code corrected:

If `vestingPeriod` is zero and fees are still locked, `feesLocked` is added to the claimable fees and `feesLocked` is set to 0.

6.4 Broken/Partial ERC165 Support

Design Medium Version 1 Code Corrected

The ERC-721 specifies that the ERC-165 interface must be implemented which defines a standard method to publish and detect what interfaces a smart contract implements.

```
function supportsInterface(bytes4 interfaceID) external view returns (bool);
```

The more derived ERC-721 contracts of Kyber Network do not overwrite this function. Hence, querying the support of the additionally implemented interfaces through `supportsInterface()` will return `false`.

Code corrected:

The issue has been addressed. Function `supportsInterface` will return true for the following interfaces.

- ERC721Enumerable
- IERC721Permit

Thus, they are considered as supported by the contract according to ERC-165.

6.5 Function `Pool.unlockPool` Reentrancy

Security | Medium | Version 1 | Code Corrected

Pools are created in a locked state and need to be unlocked first. The `unlockPool` function first removes the lock and then perform the `mintCallback`. The `_initPoolStorage` is called after the callback. This is an important function that finalizes the setup of storage for the pool. This `mintCallback` after unlock and before `_initPoolStorage` can be misused by the malicious parties, since all pool functions will be available during the call. Attacker can potentially misconfigure or abuse intermediate state inconsistency for its own profit. In addition, the `mintCallback` is usually performed to whitelisted position managers, while in this case any contract can be called.

Code corrected:

The callback has been removed for unlocking pools. Now, funds have to be transferred to the pool before unlocking the pool.

6.6 Function `ERC721Permit.permit` Payable

Design | Low | Version 1 | Code Corrected

The function `permit` has a payable modifier while abstract class ERC721Permit does not have any other functions that can withdraw funds. The BasePositionManager that inherits this class has a separate `receive` function for ether transfers. Hence, the `payable` modifier could be removed from `permit`.

Code corrected:

The payable modifier was removed from the `permit` function.

6.7 Function `Pool.burnRTokens` Natspec

Design | Low | Version 1 | Specification Changed

The `burnRTokens` does not describe the `bool isLogicalBurn` argument with a `@param` tag. This argument greatly influences the result of burn and thus should be described.

Specification changed:

isLogicalBurn is now documented.

6.8 Function Pool.burnRTokens Potential Reentrancy

Design | Low | Version 1 | Code Corrected

Certain ERC20 tokens perform callback on token transfers. For example, ERC777. Performing _burn after transfers is then can be recognized as a reentrancy pattern. While the burnRTokens and other Pool contract functions have reentrancy lock protection, there is possibility, that external contracts called during the transfer callback, might misinterpret the State of the Pool contract. For example, the reinvestL / totalSupply ratio will be off during this callback.

```
if (tokenQty > 0) token0.safeTransfer(msg.sender, tokenQty);
tokenQty = QtyDeltaMath.getQty1FromBurnRTokens(sqrtP, deltaL);
if (tokenQty > 0) token1.safeTransfer(msg.sender, tokenQty);

_burn(msg.sender, _qty);
```

Code corrected:

The transfers have been moved to the very end of the function.

6.9 Function SwapMath.calcFinalPrice Rounding Down

Correctness | Low | Version 1 | Code Corrected

The calcFinalPrice calculates the final price for swaps, when the used amount hits the specified amount limit. Depending on the starting price and direction of price movement during the swap, the price needs to be rounded either up or down. If isToken0 == false && isExactInput == true, sqrtP increases and thus price needs to be rounded down, in order not to 'overshoot' the target price.

But the tmp component of the final price is computed with rounding up division operator:

```
uint256 tmp = FullMath.mulDivCeiling(absDelta, C.TWO_POW_96, currentSqrtP);
return FullMath.mulDivFloor(liquidity + tmp, currentSqrtP, liquidity + deltaL);
```

Thus the returned value with certain chance will be more than intended.

Code corrected:

The code has been adjusted such that now division is rounding down.

6.10 Gas Inefficiency in insert()

Design | Low | Version 1 | Code Corrected

Insertions into the linked list through `LinkedList:insert()` occur only in internal function `PoolTicksState:_updateTickList`. In `insert()` the following storage read occurs:

However, that value corresponds to the last `nextTick` in `_updateTickList`. Thus, storage reads could be reduced by passing an additional argument to `insert`.

Code corrected:

`insert` now takes `nextTick` as an additional argument, reducing the number of storage reads.

6.11 Pool `swap` Max Tick Distance

Correctness Low Version 1 Code Corrected

In the main loop of the swap function, to ensure that the `tickOutside` value is interpreted correctly the `currentTick` variable needs to be adjusted if the swap moves the price down:

```
swapData.currentTick = willUpTick ? tempNextTick : tempNextTick - 1;
```

On the next iteration of the loop, the new target tick distance should not exceed the `MAX_TICK_DISTANCE == 487`:

```
int24 tempNextTick = swapData.nextTick;
if (willUpTick && tempNextTick > C.MAX_TICK_DISTANCE + swapData.currentTick) {
    tempNextTick = swapData.currentTick + C.MAX_TICK_DISTANCE;
} else if (!willUpTick && tempNextTick < swapData.currentTick - C.MAX_TICK_DISTANCE) {
    tempNextTick = swapData.currentTick - C.MAX_TICK_DISTANCE;
}
```

If `willUpTick == false` and `tempNextTick - 1`, then the `tempNextTick` will have at most 488 ticks between the matching tick for `sqrtP`. Thus, desired `Δx*fee / x < 0.0005` ratio can be violated.

Code corrected:

The `MAX_TICK_DISTANCE` was changed to 480. This way the desired `Δx*fee / x < 0.0005` ratio will be preserved.

6.12 Position Manager Storage Access

Design Low Version 1 Code Corrected

AntiSnipAttackPositionManager and BasePositionManager often read same fields inside `pos` storage variable multiple times during the function execution. Since this struct type variable is defined as a storage one, this will lead to repeated reads from the same work. More efficient approach would be utilization of in memory variables.

```
Position storage pos = _positions[params.tokenId];
```

Code corrected:

In version 3 of the code the gas is saved by utilizing memory variable for data access during the execution.

6.13 Solidity Compiler Pragma

Design | Low | Version 1 | Code Corrected

The smart contracts inside the repository utilize different compiler pragmas:

- pragma solidity >=0.5.0;
- pragma solidity >=0.8.0;
- pragma solidity ^0.8.0;
- pragma solidity >=0.8.0 <0.9.0;
- pragma solidity 0.8.9;

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively. In addition, fixed pragma ensures that the testing and deployment performed on code that was compiled by the same compiler version.

Code corrected:

Core and periphery contracts use now `pragma solidity 0.8.9` while libraries use `>=0.8.0`.

6.14 Specification Mismatches in SwapMath

Correctness | Low | Version 1 | Specification Changed

Some mismatches between the specifications and code occur in the SwapMath library. Some examples are:

- The Core Library Swap Math documentation of `calcReachAmount()` distinguishes four cases. However, case 1 & 4 and case 2 & 3 are identical. That mismatches the technical documentation of the swap and the implementation.
- The technical documentation does not specify that the absolute value of `usedAmount` (delta x tmp) is to be used for the calculation of `deltaL`.
- The technical documentation differs in the mathematical formula for calculating `returnedAmount`.

Specification changed:

The specification now better reflects the implementation.

6.15 `flash()` Sends Fees to `feeTo`

Correctness | Low | Version 1 | Specification Changed

The natspec documentation of `flash()` in IPoolActions specifies the following:

```
/// @dev Fees collected are distributed to all rToken holders
/// since no rTokens are minted from it
```

However, the fees are transferred to the `feeTo` address stored in the Factory contract.

Specification changed:

The natspec specification has changed to specify that `feeTo` receives the fees from the flash loan.

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 Pools for Tokens With Multiple Addresses

Note **Version 1**

The factory creates pools for two token address. It reverts if either the two addresses are identical or the pool has been already initialized for the token pair and the fee. However, some tokens (e.g. TUSD) have two addresses for the token. That allows for the creation of TUSD / TUSD pools, and multiple TUSD / other token pools with the same fee.