STATE MIND

Valantis Core

14-02-2024 – 19-03-2024

Table of contents

1. Project Brief

Title	Description
Client	Valantis Labs
Project name	Valantis Core
Timeline	14-02-2024 - 19-03-2024
Initial commit	bcc78b83fb44194e6e8d85f2dacf7aaa50a46553
Final commit	4c04b48913f1288f5c5f94484d229ae39d97948f

Short Overview

Valantis is DeFi protocol which takes a general and flexible approach towards core DEX design. Valantis pools contain highly reusable modules, each having a specific role such as dynamic swap fees (Swap Fee Module), value-exchange and LP position logic (Liquidity Module), pool data accumulators (Oracle Module), custom access management conditions (Verifier Module), and even decide where pool reserves should be kept by default (e.g. pool itself or custom vault).

The Universal pool decouples order-flow and LP algorithms. The user only has to interact with a single pool contract, but on the other side different classes of LPs can plug-and-play their liquidity modules to this pool, according to their custom requirements. This enables LPs to innovate on their strategies quickly, and still be able to tap into uninterrupted order-flow.

Sovereign Pool is a smart contract which allows one to pick any combination of design choices, while keeping the gas overhead and implementation complexity as low as possible. It contains the following key components:

- Liquidity Module
- Swap Fee Module (Optional)
- Oracle Module (Optional)
- Verifier Module (Optional)
- Sovereign Vault (Optional)
- Gauge (Optional)
- Pool Manager (Optional)

The goal of Sovereign Pools is to allow developers to deploy new DEX designs while writing the minimum amount of code, by re-using existing modules and core pool functionality as a starting point.

Project Scope

The audit covered the following files:

- UniversalPoolFactoryHelper.sol
- StateLib.sol
- GM.sol
- ALMLib.sol
- UniversalPool.sol
- SovereignPool.sol
- UniversalPoolStructs.sol
- SovereignPoolStructs.sol
- ReentrancyGuardStructs.sol
- UniversalPoolFactory.sol
- SovereignPoolFactory.sol
- PriceTickMath.sol
- EnumerableALMMap.sol
- ReentrancyGuard.sol
- Constants.sol
- UniversalPoolReentrancyGuard.sol
- UniversalALMStructs.sol
- SovereignALMStructs.sol
- ConstantSwapFeeModuleFactory.sol
- ConstantSwapFeeModule.sol
- ProtocolFactory.sol

2. Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	0 (0 fixed, 0 acknowledged)
Medium	5 (3 fixed, 2 acknowledged)
Informational	32 (26 fixed, 6 acknowledged)
Total	37 (29 fixed, 8 acknowledged)

4. Conclusion

During the audit of the codebase, 37 issues were found in total:

- 5 medium severity issues (3 fixed, 2 acknowledged)
- 32 informational severity issues (26 fixed, 6 acknowledged)

The final reviewed commit is 4c04b48913f1288f5c5f94484d229ae39d97948f

5. Findings report

MEDIUM-01	When performing multi-hop swap intermediary exchanges between pool tokens, they aren't reported to the **SovereignOracle**	Fixed at 1ab624

Description

Line: SovereignPool.sol#L771

The **SovereignOracle** can only perform accounting on the pair of tokens with which the pool was initialized. However, when an external **SovereignVault** is used, ALM can perform multi-hop swaps using one of the tokens as intermediaries.

Swap results are only reported to the SovereignOracle if the trade is between the pool's tokens: **_swapParams.swapTokenOut == address(swapCache.tokenOutPool)**. However, the multi-hop swap internally uses both pool tokens as intermediaries but doesn't report it to the oracle.

Recommendation

One potential solution would be disallowing multi-hop swaps when the **SovereignOracle** is set, because this action can't be undone and will lead to potentially blocking some swaps, necessary UI warnings should be put in place.

Alternatively, prohibit setting the **SovereignOracle** if external **SovereignVault** is used. This is rather restrictive, so another option we recommend is exploring specifying whether or not multi-hop swaps will be supported by a **SovereignPool** inside the constructor as an immutable variable.

MEDIUM-02	Reorg attack on creating a pool	Fixed at ae27be

Description

Lines:

- UniversalPoolFactory.sol#L29
- SovereignPoolFactory.sol#L29

The **SovereignPoolFactory** and **UniversalPoolFactory** use **create2** which allows specifying salt to prevent an attack. During the reorg attack, the network forks and has a different network state in each of the forks, and when a consensus is reached, one of the forks is dropped. Thus, it is important for the deployer that the contract address is changed if the contract configuration parameters have been changed. The current implementations allow an attacker to create pools with different configurations in every fork. Therefore, the victim can deposit funds into a pool with an unexpected configuration / malicious pool.

The scenario:

1. Alice sends a transaction to deploy a **SovereignPool** and an ALM.
2. Bob sees Alice's transaction and sends a transaction to deploy a **SovereignPool** with different constructor args and the same ALM.
3. Alice's transaction is confirmed and **SovereignPool** is deployed at **poolAddress**.
4. Bob's transaction is confirmed and **SovereignPool** is deployed at **poolAddress2**.
5. Alice sees that the pool has been deployed with her arguments and sends a transaction to deposit funds to the pool through the ALM.
6. A reorg occurs and now Bob's transaction comes before Alice's transaction and Bob's pool is deployed at **poolAddress** and Alice's pool is at **poolAddress2**.
7. Alice's transaction to deploy funds is confirmed and her funds are sent to Bob's pool.
8. Bob can then use the verifier module to block Alice from withdrawing her funds.

While reorgs on Ethereum are mostly of depth 1, on an L2 like Polygon reorgs depth can be very high.

Impact: Getting stuck or losing deposited funds

Recommendation

We recommend adding the pool constructor arguments for the salt creation

MEDIUM-03	Loss of pool manager's commissions	Acknowledged

Description

Lines:

- SovereignPool.sol#L955
- SovereignPool.sol#L969-L970

In **SovereignPool._claimPoolManagerFees** function, the pool calls **sovereignVault** to collect commissions using **ISovereignVaultMinimal.claimPoolManagerFees**. However, current checks of the transferred amount allow **sovereignVault** not to transfer the requested amount in total. The next step is to reset the value of **feePoolManager0**, and **feePoolManager1**. Thus, if the **sovereignVault** didn't transfer the full amount during the call, the pool will not be able to request the balance again.

Recommendation

We recommend returning additional information from the **ISovereignVaultMinimal.claimPoolManagerFees** call so that the pool can multiple various commission accounting scenarios.
Possible scenarios:

- **sovereignVault** decides to give only part of the fees regardless of the requested amount, therefore zeroing **feePoolManager0**, **feePoolManager1** is necessary
- **sovereignVault** can't withdraw all fees in the current call. The caller should make another call later and reset the **feePoolManager0**, **feePoolManager1** is not necessary

Client's comments

In case fees sent by sovereignVault are not in sync with the pool's storage variables, sovereignVault is a faulty implementation. Our approach is to discourage these kinds of scenarios by zero-ing out the fees in case there is a mismatch. The poolManager should just deploy a new pool with a correct implementation of sovereignVault. Additionally, allowing leftover fees may open up manipulation of VAL emission calculations for Sovereign Gauges.

MEDIUM-04	Rounding in favor of the protocol account for the last token in a quote by incorrect price.	Acknowledged

Description

Lines:

- GM.sol#L414-L418

Let's look **PriceTickMath.getTokenInAmount()** on examples:

```
Price Tick = 100000 => 1 : 22015
tokenOutAmount = 0 => tokenInAmount = 0
tokenOutAmount = 1 => tokenInAmount = 1 - Incorrect because price 1 : 22015
tokenOutAmount = 22015 => tokenInAmount = 1
tokenOutAmount = 22016 => tokenInAmount = 2
```

Therefore, using the function to calculate the number of filled order tokens may lead to incorrect filling of the last token during the processing of each quote. This allows ALM to exchange the last token at 1:1. This, it is always profitable for ALM to provide inaccurate quotes with 1 extra wei to fill the last token at 1:1, regardless of **currentPriceTick**. Also, a swapper can make a 1-to-1 swap regardless of the **spotPriceTick** installed in the pool. An attacker can also make a 1-to-1 swap to affect the work of the oracle because the amounts of the actual exchange differ from the spot price.
POC: **UniversalPool_POC.test_POC_incorrect_filling**

Recommendation

We recommended rounding down for **PriceTickMath.getTokenInAmount()** because the current implementation already uses the inaccuracy of calculating the price in favor of LP.

Client's comments

> In our understanding the risk from this kind of rounding, is isolated to possible oracle manipulation, and general swapping logic is sound. If this is the case, then for all oracle docs we will add a warning that this case is possible. Changing PriceTickMath.getTokenInAmount to round down would break one of the invariants outlined in the formal verification work, as well as causing significant changes. We will mention about these scenarios in the docs, so that ALM, oracle developers and swap users are aware.

MEDIUM-05	Reorg attack on contract creation	Fixed at f02f30

Description

Lines:

1. ProtocolFactory.sol#L542
2. ProtocolFactory.sol#L575
3. ProtocolFactory.sol#L607
4. ProtocolFactory.sol#L947

The **ProtocolFactory** contract deploys different contracts using the create2.
The **ProtocolFactory** uses create2 which allows specifying salt to prevent an attack. During the reorg attack, the network forks have a different network state in each of the forks, and when a consensus is reached, one of the forks is dropped. Thus, it is important for the deployer that the contract address is changed if the contract configuration parameters have been changed. The current implementations allow an attacker to create modules with malicious configurations in each fork. Therefore, the attacker can set himself up as a manager for these modules.
The scenario:

1. Alice calls **deploySwapFeeModuleForPool** to deploy a **swapFeeModule**.
2. Bob sees Alice's transaction and sends a transaction to deploy **swapFeeModule** with **_swapFeeBips** bigger than Alice's **swapFeeModule**.
3. Alice's transaction is confirmed and **swapFeeModule** is deployed at **swapFeeModuleAddress1**.
4. Bob's transaction is confirmed and **swapFeeModule** is deployed at **swapFeeModuleAddress2**.
5. Alice sees that the module has been deployed with her arguments and sends a transaction that sets **swapFeeModule** in the pool with **swapFeeModuleAddress1**.
6. A reorg occurs and now Bob's transaction comes before Alice's transaction and Bob's module is deployed at **swapFeeModuleAddress1** and Alice's module is at **swapFeeModuleAddress2**.
7. Alice's transaction to set **swapFeeModule** is confirmed and her pool takes more fee because she sent the transaction with Bob's module address. She can change **swapFeeModule** only after 3 days.

This also can happen with the oracle module and ALM.
While reorgs on Ethereum are mostly of depth 1, on an L2 like Polygon reorgs depth can be very high.
Impact: Users will pay more fees, Alice will not be able to change the oracle module, and malicious ALM will be added to a pool.

Recommendation

We recommend adding **msg.sender** and constructor arguments for the salt creation.

INFORMATIONAL-01	Resetting PoolManager resets fees, but doesn't emit the corresponding event	Fixed at ae27be

Description

Line: SovereignPool.sol#L451
Setting pool manager to **address(0)** also sets **poolManagerFeeBips = 0**, but doesn't emit the corresponding event **PoolManagerFeeSet**.

Recommendation

We recommend emitting **PoolManagerFeeSet(0)** event when new manager address is **address(0)**.

INFORMATIONAL-02	If no swapFeeModule is set, check for MAX_SWAP_FEE_BIPS is unnecessary	Fixed at ae27be

Description

Lines:

- SovereignPool.sol#L688-L699
- UniversalPool.sol#L521-L532

If **swapFeeModule** is set, the call **swapFeeModule.getSwapFeeInBips()** is made. The return value, **feeInBips**, is then checked to ensure it does not exceed **_MAX_SWAP_FEE_BIPS**. However, if **swapFeeModule** isn't set, the **defaultSwapFeeBips** is used. It is guaranteed in SovereignPool.sol#L317-L319 and UniversalPool.sol#L193 that **defaultSwapFeeBips** does not exceed **_MAX_SWAP_FEE_BIPS**, rendering the check in the second case redundant.

Recommendation

We recommend moving the **_MAX_SWAP_FEE_BIPS** check into a separate code block:

```
SwapFeeModuleData memory swapFeeModuleData;

if (address(swapCache.swapFeeModule) != address(0)) {
    swapFeeModuleData = swapCache.swapFeeModule.getSwapFeeInBips(
        _swapParams.isZeroToOne,
        _swapParams.amountIn,
        msg.sender,
        _swapParams.swapContext.swapFeeModuleContext
    );
    if (swapFeeModuleData.feeInBips > _MAX_SWAP_FEE_BIPS) {
        revert SovereignPool__swap_excessiveSwapFee();
    }
} else {
    swapFeeModuleData = SwapFeeModuleData({ feeInBips: defaultSwapFeeBips, internalContext: new bytes(0) });
}
```

INFORMATIONAL-03	Redundant check when writing an oracle update, amountOut is always above 0	Fixed at ae27be

Description

Line: SovereignPool.sol#L773

The check **amountOut > 0** is redundant, as it is already performed prior:

```
if (amountOut == 0) {
    return (0, 0);
}
```

Recommendation

We recommend removing this check.

INFORMATIONAL-04	SovereignPool.withdrawLiquidity() must be blocked when funds are stored in SovereignVault	Fixed at 1ab624

Description

Line: SovereignPool.sol#L883

When a **SovereignVault** is set to external contract, there is no point in allowing ALM to call **SovereignPool.withdrawLiquidity()**.

Recommendation

We recommend adding a check:

```
if (sovereignVault != address(this)) {
  revert SovereignPool__withdrawLiquidity_withdrawDisabled();
}
```

Similiar to the one in **SovereignPool.depositLiquidity()**.

INFORMATIONAL-05	An unnecessary external call when checking balances	Fixed at ae27be

Description

Line: SovereignPool.sol#L828

The **SovereignPool.depositLiquidity** function checks the balance delta after the user deposits tokens but has an extra external call to the verification module. The general rule for delta checks is to get the balances before and after the callback. Any transfer of tokens from the verification module to the pool will be considered as a user deposit. This scenario is unlikely, but it is wrong.

Recommendation

We recommend getting token pre-balances after the verification module call.

INFORMATIONAL-06	Deposited tokens are not counted for zero input amounts	Fixed at ae27be

Description

Lines:

- SovereignPool.sol#L838
- SovereignPool.sol#L856

SovereignPool.depositLiquidity function checks the user deposited amounts SovereignPool.sol#L844-L846, SovereignPool.sol#L848, SovereignPool.sol#L862-L864, SovereignPool.sol#L866. However, if **_amount0 == 0 && amount0Deposited > 0** or **_amount1 == 0 && amount1Deposited > 0** non-rebase tokens will not be counted on the balance of reserves and the call doesn't revert.

Impact: loss of funds transferred by mistake

Recommendation

We recommended adding a check for this scenario

```
if (_amount != 0) {
      ...
} else if (amountDeposited > 0) {
      revert SovereignPool__depositLiquidity_incorrectTokenAmount();
}
```

INFORMATIONAL-07	The delta balance check can be bypassed in future **sovereignVault** implementations	Acknowledged

Description

Lines:

- SovereignPool.sol#L1030-L1033

The **sovereignVault** can be set as an external contract to a pool that receives input and transfers output amounts. The current architecture tries to be flexible. Vault implementations will have the functions of depositing and withdrawing liquidity (this may be available to users or a separate Vault manager role, or managed in another way). However, it opens the possibility of a recoverable transfer attack during the swap.

Let's look at a potential scenario:

1. The attacker swaps the pool
2. From callback **ISovereignPoolSwapCallback.sovereignPoolSwapCallback** calls **SovereignVault.deposit** that transfer tokens from the attacker and saves the deposited balance in Vault
3. Checking the balance delta is successful in the pool contract
4. The attacker withdraws calls **SovereignVault.withdraw** that return deposited tokens from Vault

Recommendation

We recommend transferring tokens using the **ERC20.transferFrom** function in the pool contract and relying on token approval in the **ISovereignPoolSwapCallback.sovereignPoolSwapCallback** callback.

Client's comments

> Relying on approval during swap callback would break composability with routers and solvers. We will mention in the docs that sovereignVault should be appropriately locked during swaps.

INFORMATIONAL-08	Logically incorrect usage _MAX_SWAP_FEE_BIPS	Fixed at 1ab624

Description

Lines:

- SovereignPool.sol#L933
- SovereignPool.sol#L963-L964
- SovereignPool.sol#L1038
- SovereignPool.sol#L1064
- SovereignPool.sol#L1079
- GM.sol#L251-L252
- GM.sol#L279
- StateLib.sol#L86
- StateLib.sol#L93-L94

_MAX_SWAP_FEE_BIPS constant has a 10_000 value, but logically incorrect to use this constant for calculating other types of fees because these calculations depend on a 100% representation in base points.

Recommendation

We recommended adding **Constants.FACTOR_ONE = 10_000** constant and using it in these calculations.

INFORMATIONAL-09	SwapFeeModule doesn't have enough information to provide the correct fee bps	Fixed at 63c643

Description

Lines: SovereignPool.sol#L689-L694
The pool allows users to make swaps **tokenOutput** can be a pool's token or another token supported by **sovereignVault**. However, the **SwapFeeModule** doesn't address the swapped tokens, only the **isZeroToOne** flag is given. Therefore, it is impossible to implement a different fee size depending on the **tokenOutput**, and the **isZeroToOne** flag doesn't provide enough information about the exchange.

Recommendation

We recommend providing **tokenInput** and **tokenOutput** in the **ISwapFeeModule.getSwapFeeInBips** call

INFORMATIONAL-10	Unnecessary checks	Fixed at ae27be

Description

Lines:

- SovereignPool.sol#L850
- SovereignPool.sol#L868

The check **if (amount0Deposited > 0)** is unnecessary since **amount0Deposited == _amount0** and **_amount0 != 0**. The same for check **if (amount1Deposited > 0)**.

Recommendation

We recommend removing unnecessary checks.

INFORMATIONAL-11	Rounding not in favor of the protocol	Fixed at ae27be

Description

Line: SovereignPool.sol#L747.
When calculating **effectiveFee** the division is rounded down, which can lead to no fee being taken for very small **amountInFilled** and **feeInBips** values. The calculation of fees should use rounding in favor of the protocol and LPs. With rounding up **amountInUsed** will not be greater than **amountIn** because **amountInFilled < amountInWithoutFee**.

Recommendation

We recommend rounding up this calculation.

INFORMATIONAL-12	Return instead of revert	Fixed at ae27be

Description

Line: **SovereignPool.sol#L740** - makes **return (0, 0)** instead of **revert()**.
However, such a check does not cancel the side effects of this exchange and means the actual execution of certain calculations. Moreover, the **ISovereignALM.getLiquidityQuote()** function modifier is not view, therefore it can change its state. Thus, without performing the actual exchange, we rewrite or change the state recorded in the ALM, which can be used for manipulation.
We believe that side effects in the absence of exchange should be annulled.

Recommendation

We recommend reverting the transaction in case there's no output amount.

INFORMATIONAL-13	Optimizations	Fixed at 579c95

Description

Lines:

- ALMLib.sol#L98-L99 - can be wrapped in an unchecked block, because of the check ALMLib.sol#L94.
- ISwapFeeModule.sol#L33-L40 - it can be replaced by using **ISwapFeeModule**. It is used only for tests.
- UniversalPoolFactory.sol#L31-L33 - enough check **deployment != address(0)** because **address(0)** if the **create2** failed. Simplifying checking will save gas and reduce the size of the bytecode.
- UniversalPool.sol#L189 - zero address check, because if **poolManager == address(0)** the pool can't be initialized UniversalPool.sol#L283
- UniversalPool.sol#L375 - implicit behavior, if **manager == 0 && poolManagerFeeBips > 0**, then the function should revert because the manager provided invalid input data.
- UniversalPool.sol#L579-L581 - already have a check UniversalPool.sol#L584.

Optimized version:

```
if (amountInUsed > 0) {
    if (amountOut == 0) {
        revert UniversalPool__swap_zeroAmountOut();
    }
    ...
}
```

Recommendation

We recommend added optimizations.

INFORMATIONAL-14	Loss of deposited tokens during the swap	Acknowledged

Description

Line: UniversalPool.sol#L546

At the beginning of the swap, the **UniversalPool** blocks **WITHDRAW** and **SWAP** locks UniversalPool.sol#L512-L513. Next, ALMs reserves are fetched to memory UniversalPool.sol#L740-L743. After the **GM.setupSwap** stage, the **UniversalPool** blocks the **DEPOSIT** lock. In the current implementation, the **UniversalPool** allows depositing to any ALM during **GM.setupSwap**. However, if **ALM_1** deposits to **ALM_2** from the **IUniversalALM.setupSwap** call, the previous reserve values will remain in the contract memory and incorrect reserve values will be written at the end of the swap GM.sol#L306-L311. Thus, the deposited tokens will be lost.

Recommendation

We recommend(one of):

- adding **almID** or **almAddress** in the **DEPOSIT** lock structure. Thus, at the beginning of the swap, the pool can block all deposits and before calling **IUniversalALM.setupSwap** unlock deposits only for the called ALM. In this way, the pool will be able to ensure that the data in memory corresponds to the reserve values.
- updating documentation for ALM and highlighting this case

Client's comments

> We want ALMs to be aware that they should not perform such cross-ALM deposits during setupSwap. We will update docs accordingly

INFORMATIONAL-15	Universal Pool.spotPriceTick() has an excessive call limit.	Fixed at 579c95

Description

Lines: UniversalPool.sol#L226-L228

UniversalPool.spotPriceTick() has a check to protect against read-only re-entry. However, most of the **UniversalPool** functions have a **nonReentrantGlobal** modifier, which is locked by a **SWAP** lock. Therefore, external contracts don't make a call to **UniversalPool.spotPriceTick()** from these functions. Protection against re-entry for reading is needed only if **state.spotPriceTick** is changed, if the functions don't change **state.spotPriceTick**, then the block is redundant because the value is immutable and this does not lead to vulnerability.

Recommendation

We recommended adding a new **Lock.SPOT_PRICE_TICK** lock(that can be locked at the beginning of the **Universal Pool.swap()** call. Also, this is optimized for gas because it can be stored in an existing lock slot. The **nonReentrantGlobal** modifier will remain unchanged, then the **UniversalPool.swap()** function can't be called from external contracts, because has an additional check by **Lock.SWAP** lock. **UniversalPool.spotPriceTick()** should have a reentrancy check by **Lock.SPOT_PRICE_TICK**.

```
function spotPriceTick() external view override returns (int24) {
    if (_poolLocks.spotPriceTick != _NOT_ENTERED) {
        revert UniversalPool__spotPriceTick_spotPriceTickLocked();
    }

    return _spotPriceTick;
}
```

INFORMATIONAL-16	Incorrect check for input amount in UniversalPool.swap()	Fixed at 579c95

Description

Line: UniversalPool.sol#L598

During the exchange, the contract receives input tokens and either calls **IUniversalPoolSwapCallback.universalPoolSwapCallback()** or calls **SafeERC20.safeTransferFrom()**. After this, the contract checks that **tokenInInterface.balanceOf(address(this)) – tokenInPreBalance < amountInUsed**. This check allows the user to send more tokens than needed and the extra tokens will be stuck in the contract.

Recommendation

We recommend changing the check to **tokenInInterface.balanceOf(address(this)) != amountInUsed + tokenInPreBalance**

INFORMATIONAL-17	Lack of zero address validation in SovereignPool.setGauge()	Fixed at 579c95

Description

Line: SovereignPool.sol#L499

SovereignPool.setGauge() function, doesn't check that **_gauge** address does not equal zero.

Recommendation

We recommend adding a zero address check for **_gauge**. Like in UniversalPool.sol#L341-L343

INFORMATIONAL-18	Rounding issue when calculating **effectiveFee**	Fixed at 5833a5

Description

Line: UniversalPool.sol#L568.

1. Similar to **SovereignPool.swap()** implementation, **effectiveFee** is not rounded up in favor of LPs.
2. In case when **amountInRemaining == 0**, the calculation introduces a rounding error in favor of the user.
3. Also, there is an optimization:

```
amountInUsed = swapCache.amountInMinusFee - swapCache.amountInRemaining;

swapCache.effectiveFee = Math.mulDiv(
        amountInUsed,
        swapFeeModuleData.feeInBips,
        MAX_SWAP_FEE_BIPS
    );

almStates.updatePoolState(_ALMPositions, _state, _swapParams, swapCache);

amountInUsed += swapCache.effectiveFee;
```

Recommendation

We recommend to change to:

```
if (swapCache.amountInRemaining != 0) {
    swapCache.effectiveFee = Math.mulDiv(
        (swapCache.amountInMinusFee - swapCache.amountInRemaining),
        swapFeeModuleData.feeInBips,
        MAX_SWAP_FEE_BIPS,
        Math.Rounding.Up
    );
} else {
    swapCache.effectiveFee = _swapParams.amountIn - swapCache.amountInMinusFee;
}
```

and implement the optimization.

INFORMATIONAL-19	Flashloan implementation in UniversalPool allows transferring excees tokens	Fixed at 579c95

Description

Line: UniversalPool.sol#L323

In function **UniversalPool.flashLoan()**, to repay the flashloan, **_receiver** should approve the amount it asked to borrow. Contract checks that the amount of tokens after **SafeERC20.safeTransferFrom()** call isn't less than before the flashloan: **flashToken.balanceOf(address(this)) < poolPreBalance**. However, this allows the **_receiver** to send extra tokens during flashloan execution which will not be accounted in the pool's reserves thus get stuck.

Recommendation

We recommend using strict check:

```
if (flashToken.balanceOf(address(this)) != poolPreBalance) {
    revert ValantisPool__flashLoan_flashLoanNotRepaid();
}
```

To be consistent with implementation in **SovereignPool.sol#L578**.

INFORMATIONAL-20	UniversalPool shouldn't allow swaps with zero amountOut	Fixed at 5833a5

Description

Lines: **UniversalPool.sol#L579-L581**.

UniversalPool.swap() allows swap even if **amountInUsed != 0** and **amountOut > 0**.

If one of the parties doesn't receive tokens, either **amountInUsed == 0** or **amountOut == 0** the swap should revert.

There is already a check in place that **amountIn** can't be zero, **UniversalPool.sol#L682-L684**, but there is no zero check for **amountOut**.

Recommendation

We recommend simplifying the check to only proceed when **amountOut** is equal to zero:

```
if (amountOut == 0) {
    revert UniversalPool__swap_zeroAmountOut();
}
```

Also you can remove duplicate check **UniversalPool.sol#L603**.

INFORMATIONAL-21	Code duplication when adding ALM to EnumerableALMMap	Fixed at 579c95

Description

Lines:

- **EnumerableALMMap.sol#L135**
- **EnumerableALMMap.sol#L143**

When a new ALM is added, it's corresponding index is updated by the code specified above. However, we know that inside one of the **if**-clauses **metaALMPointer == almLength** and there is no need to differentiate between those two variables, thus saving GAS.

Recommendation

We recommend moving the code of setting ALM index outside of conditional blocks:

```
_set._indexes[_alm.slot0.almAddress] = metaALMPointer + 1;
```

INFORMATIONAL-22	Some little logical/gas improvements	Fixed at 5833a5

Description

Lines:

- **UniversalPool.sol#L164** - modifier **onlyActiveALM** is used only once inside **UniversalPool** contract. Furthermore, an identical result check may be inside the **ALMLib** library. We can add a check here, leading to the same result, cause **getALM** function already returns the status of ALM. As a result, we will remove unused modifier, error, and internal function inside **UniversalPool**, and optimize this check by gas consumption. Furthermore, external function **EnumerableALMMap.isALMActive()** can be removed.

  ```
  (ALMStatus status, ALMPosition storage almPosition) = _ALMPositions.getALM(msg.sender);

  if (status != ALMStatus.ACTIVE) {
     revert ALMLib__depositLiquidity_onlyActiveALMCanDeposit();
  }
  ```

- **UniversalPool.sol#372** - function **UniversalPool.setPoolState** uses as argument **_newState** variable with **PoolState** struct, but it uses only 4 variables from here. We recommend replacing this function with several separate functions, which will change **swapFeeModule**, **PoolManager**, **universalOracle**, and **poolManagerFees** accordingly.
- **GM.sol#L453** - function **_getContext** is getting **almState** variable, but uses only one bool flag from that. It can be optimized by sending only this flag instead of the full struct.

Recommendation

We recommend making the improvements listed.

INFORMATIONAL-23	Some gas optimizations	Acknowledged

Description

Lines:

- **GM.sol#L77** – here a **setupQuote** variable is used as a result of **UniversalALM.setupSwap** function, but it's an almost full copy of **InternalSwapALMState.latestLiquidityQuote** variable. The difference between these variables is only in the **priceTick** variable, which may used only in **IUniversalALM.callbackOnSwapEnd()**. Furthermore, you are rewriting the result of **setupQuote** to **InternalSwapALMState.latestLiquidityQuote** here and here.
- We recommend moving the **ALMLiquidityQuote** struct inside **InternalSwapALMState.ALMCachedLiquidityQuote**. So, you can remove **tokenOutAmount**, **nextLiquidPriceTick** and **internalContext**, add **ALMLiquidityQuote** here and move **priceTick** up/down.

```
struct ALMCachedLiquidityQuote {
    ALMLiquidityQuote almLiquidityQuote;
    int24 priceTick;
}
```

- Now you can remove variable creation **GM.sol#L75**, and save the result in **InternalSwapALMState.latestLiquidityQuote**.

```
(almStates[almNum].isParticipatingInSwap, almStates[almNum].refreshReserves,
InternalSwapALMState.latestLiquidityQuote.almLiquidityQuote) =
    IUniversalALM(almStates[almNum].almSlot0.almAddress).setupSwap(
        ...
    );
```

- It will also be necessary to slightly rewrite the code associated with checking these variables.
- If ALM does not participate in the exchange (**isParticipatingInSwap == false**), then you can return empty values because the memory variables are already allocated anyway. In this case, you can add three additional checks or omit them; otherwise, these further values should not affect anything.
- **GM.sol#L166** – this line can be moved outside **while** cycle, cause number of active ALM's doesn't change inside this function.

Recommendation

We recommend making the improvements listed.

Client's comments

> In order to minimize code changes and risk affecting the formal verification work already done, we will keep as is.

INFORMATIONAL–24	Code style for future public release/audits	Fixed at 5833a5

Description

Lines:

- **EnumerableALMMap.sol#L155** – we recommend renaming informative error message to **EnumerableALMMap__addALMPosition_almAlreadyExist**.
- **EnumerableALMMap.sol#L200** – we recommend renaming informative error message to **EnumerableALMMap__removeALMPosition_almNotFound**.
- **ReentrancyGuardStructs.sol#L22** – outdated comment.
- **UniversalPool.sol#L285** – outdated comment.
- **UniversalPool.sol#L508** – outdated comment.
- **GM.sol#L405** – outdated comment.
- **StateLib.sol#L14–15** – these functions were moved to **ALMLib**. We recommend deleting these comments.

Recommendation

We recommend fixing these issues.

INFORMATIONAL–25	Gas optimization when amountOutExpected == 0	Acknowledged

Description

Line: GM.sol#L472.
If **amountOutExpected == 0**, then there is no reason to call ALMs for quotes because they can only return **amountOut == 0** because of the check at line GM.sol#L363. It can occur for small **amountInRemaining** at some price ticks. When it happens, there is no need to call ALMs in **GM.setupSwaps()** and **GM.requestForQuotes()**.
Also, at line GM.sol#L225 there is no need to make a further round of calls if **amountOutExpected == 0**.

Recommendation

We recommend checking if **amountOutExpected == 0** after computing it and skip some calls.

Client's comments

> We always prefer to round in favour of LPs. In this case if amountInRemaining is some small value, we still make the getLiquidityQuote call so that the next ALM receives the amountInRemaining (even if it is for an amountOut = 0)
> Only one extra call will be made, because whenever such a case arises, the remaining amountIn is attributed to the next ALM in the ordering.
> For simplicity, and to avoid breaking assumptions that ALMs might have about the UP, we think we should not skip the calls for amountOutExpected == 0.

INFORMATIONAL-26	Oracle arguments may be incomplete or redundant	Fixed at b48d9e

Description

Line: **UniversalPool.sol#L619**

- the arguments that are passed to the Oracle may not be sufficient for a fully functional price report.

For example, **swapCache.spotPriceTickStart** may be useful if a certain oracle implementation does not store the value of the last exchange.

Let's say we have an oracle that displays information taken from several pools with different commissions. Or which is used with dynamic **swapFeeModule**. Let's say one of our ALMs has much liquidity on one tick. Then, with different exchanges within one tick and with different commissions, the price of the oracle may change, which is not valid behavior.

We suggest reflecting on this in the documentation if you think **swapCache.effectiveFee** can be used in certain cases. This also applies to the **swapCache.limitPriceTick** parameter, which reflects only the user's preferences to receive more tokens from the exchange, but not the real price of the asset.

Recommendation

We recommend removing **swapCache.limitPriceTick** parameter and adding a **swapCache.spotPriceTickStart** one.

We also recommend adding information about **swapCache.effectiveFee** to the documentation.

INFORMATIONAL-27	Unused and redundant code	Fixed at e1e526

Description

Lines:

- ProtocolFactory.sol#L26
- ProtocolFactory.sol#L28
- ProtocolFactory.sol#L30
- ProtocolFactory.sol#L32
- ProtocolFactory.sol#L39
- ProtocolFactory.sol#L48
- ProtocolFactory.sol#L50
- ProtocolFactory.sol#L57 – errors are not used and can be removed
- ProtocolFactory.sol#L407
- ProtocolFactory.sol#L673
- ProtocolFactory.sol#L681
- ProtocolFactory.sol#L688
- ProtocolFactory.sol#L690
- ProtocolFactory.sol#L767
- ProtocolFactory.sol#L775
- ProtocolFactory.sol#L782
- ProtocolFactory.sol#L784 – redundant address conversion, because storage variables already have address type.
- ProtocolFactory.sol#L918 – **EnumerableSet.add** returns a bool that can be used instead of **EnumerableSet.contains**. Details in OZ docs
- ProtocolFactory.sol#L930 – **EnumerableSet.remove** returns a bool that can be used instead of **EnumerableSet.contains** . Details in OZ docs
- ConstantSwapFeeModule.sol#L121
- ConstantSwapFeeModule.sol#L127 – the functions does nothing, so access restriction is not needed, the **onlyPool** modifier can be removed. Also, pools call **ISwapFeeModule.callbackOnSwapEnd()** only if the **internalContext.length > 0**.

Recommendation

We recommend removing redundant and unused code

INFORMATIONAL-28	BLOCK_TIME can be changed in future	Fixed at 4a4ce7

Description

Line: ProtocolFactory.sol#L72

ProtocolFactory stores an immutable **BLOCK_TIME** value ProtocolFactory.sol#L279 and can't be changed in future. However, the block time may be changed. Each blockchain seeks to reduce the block time to increase throughput. Thus, the values calculated based on the block time may be irrelevant after changing the block time. For example, the average Ethereum block time decreased from 17 to 12 seconds Also, the block generation time may be undetermined if the protocol is deployed in L2 and the centralized sequencer shutdown.

During the last audit, you used **BLOCK_TIME** to determine time intervals and control points. We do not recommend using **BLOCK_TIME** because the block time may be changed in the future.

Recommendation

We recommend removing **BLOCK_TIME** from **ProtocolFactory** and using timestamps for calculations time intervals and control timestamps.

INFORMATIONAL-29	Incorrect custom error in functions	Fixed at c87d0e

Description

Lines:

1. ProtocolFactory.sol#L665
2. ProtocolFactory.sol#L759

Functions **ProcolFactory.deployUniversalGauge()** and **ProtocolFactory.deploySovereignGauge()** revert with an unobvious custom error if a pool doesn't have **poolManager** and **msg.sender != protocolManager**.

Recommendation

We recommend changing the custom error **ProtocolFactory__deployUniversalGauge_onlyPoolManager()** to **ProtocolFactory__onlyProtocolManager()**.

INFORMATIONAL-30	Inconsistency of the proof of approximation and implementation of the algorithm	Acknowledged

Description

During the last audit, we provided mathematical proof of the approximation of the algorithm for finding the price and tick for the classical implementation as UniswapV3. The new implementation has a different set of magic numbers and forced rounding, which finally affects the inaccuracy and approximation. For this reason, the **PriceTickMath.getTickAtPriceOver()** function doesn't work with prices obtained using **PriceTickMath.getPriceAtTickUnder()**.

Recommendation

We recommend improving the approximation proof for the new algorithm version and implementing **PriceTickMath.getTickAtPriceUnder()** function.

Client's comments

The getTickAtPrice functions are non-essential. We will make it clear in the docs and natspec that getTickAtPriceOver only works for over-estimated prices, and will develop a more encompassing implementation once new modules request it.

INFORMATIONAL–31	Name recommendations for arguments inside external/internal libraries	Acknowledged

Description

Libraries that have different variable names:

- **EnumerableALMMap.sol**
- **GM.sol**

The general recommendation is to give the same variable the same name. This will make it easier to track changes in their status.

- Example: **GM.sol** In this library, you use the same memory variables, which can change in internal calls. However, there are quite a lot of ones and in many ways they are similar, but their names differ depending on the functions. For example: **setupQuote** and **almLiquidityQuote**, **almStates** and **internalSwapALMState**. In some cases, you change **storage** variables that are also named differently in different places.

We recommend renaming variables to the same style in all libraries/contracts, making it easier to understand what is being passed and changed within functions. If one variable is passed from the array, the difference will be **internalSwapALMStates[]** – **internalSwapALMState**. This will greatly improve not only the readability of the code but also reduce the amount of confusion in the future.

Recommendation

We recommend renaming these variables.

Client's comments

> In order to minimize code changes and risk affecting the formal verification work already done, we will keep as is and clarify variable naming in the docs and comments.

INFORMATIONAL–32	Unnecessary writing to storage	Fixed at 8f605b

Description

Lines:

- ProtocolFactory.sol#L642
- ProtocolFactory.sol#L736

The storage writes **_universalPools[_token1][_token0].add(pool)** and **_sovereignPools[args.token1][args.token0].add(pool)** are unnecessary because the functions ProtocolFactory._isValidUniversalPool() and ProtocolFactory._isValidSovereignPool() get **token0** and **token1** from the pool and check the mappings in order **token0**, **token1**. So it is sufficient to only use **_universalPools[_token0][_token1].add(pool)** and **_sovereignPools[args.token0][args.token1].add(pool)**.

On the other hand, it would be useful to add pools in reverse order of tokens if there was an external getter function. Then the users can specify tokens in any order and get the pools deployed by **ProtocolFactory**.

Recommendation

We recommend removing unnecessary storage writes or adding an external getter function for Universal and Sovereign pools deployed by **ProtocolFactory**.

STATE
MIND