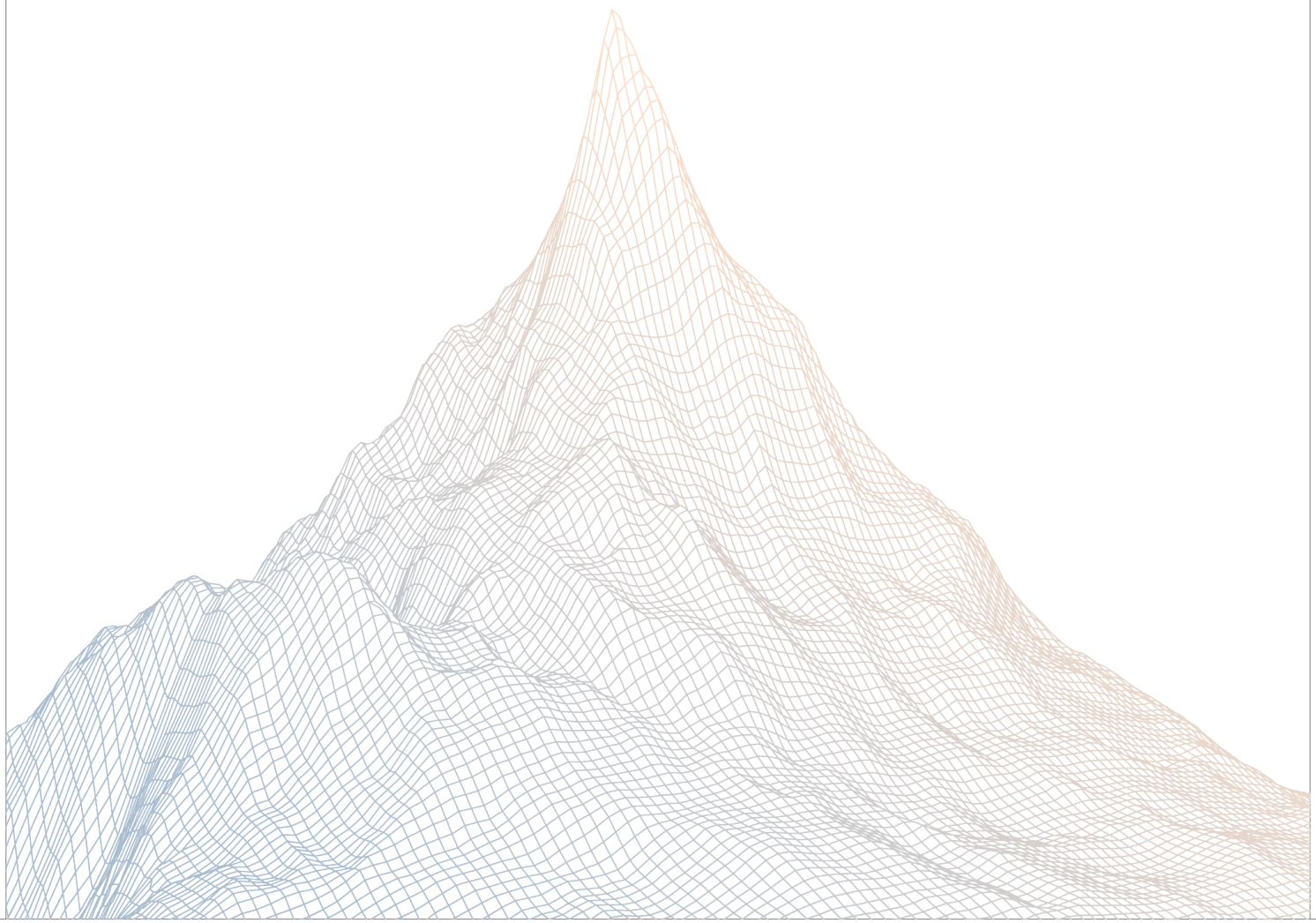
Zenith

Valantis

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 25th to March 28th, 2025

AUDITED BY: cccz
said

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Valantis

Valantis is a novel approach to representing Value-Exchange Logic, Valantis Core Pools can recover the entire DEX space using composable modules. It attempts to be a full generalization of smart-contract DEXes, enforcing strict security assumptions over the interactions between modules and users to create the most secure, composable, and developer friendly environment for DEX development. The Valantis Sovereign Pools integrate various components responsible for functions such as pricing logic, fee calculation, oracle services, reserve vaults, and liquidity management strategies. Sovereign Pools natively support rebase tokens.

2.2 Scope

The engagement involved a review of the following targets:

Target	valantis-stex
Repository	https://github.com/ValantisLabs/valantis-stex
Commit Hash	`69aa644c316859802800db0a4195f609a8bdad9a`

Files	`STEXAMM`
	`stHYPEWithdrawalModule`
	`AaveLendingModule`
	`structs`

2.3 Audit Timeline

March 25, 2025	Audit start
March 28, 2025	Audit end
March 31, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	2
Medium Risk	3
Low Risk	1
Informational	0
Total Issues	**6**

3

Findings Summary

ID	Description	Status
H-1	Incorrect poolManagerFee base when calculating instant withdrawal fees	Resolved
H-2	Deposit and withdraw doesn't consider excess native tokens in withdrawalModule	Resolved
M-1	Instant withdrawal could fail if most of the token1 are provided to the lending pool	Resolved
M-2	Inaccurate amount1SwapEquivalent when an instant withdrawal is performed	Resolved
M-3	Overcharging of instant withdrawal fees	Resolved
L-1	claim should trigger update to prevent reverts	Acknowledged

4

Findings

4.1 High Risk

A total of 2 high risk findings were identified.

[H-1] Incorrect poolManagerFee base when calculating instant withdrawal fees

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- STEXAMM.sol#L584-L591

Description:

In SovereignPool, `poolManagerFeeBips` refers to the percentage of the `poolManagerFee` out of all `swapFees`, it has a maximum of 50%. `swapFee` is just a small portion of the input tokens.

```
        swapCache.amountInWithoutFee = Math.mulDiv(
            _swapParams.amountIn,
            _MAX_SWAP_FEE_BIPS,
            _MAX_SWAP_FEE_BIPS + swapFeeModuleData.feeInBips
        );
...
            effectiveFee = _swapParams.amountIn
    - swapCache.amountInWithoutFee;
...
        if (isTokenInRebase && sovereignVault == address(this) &&
    poolManager != address(0)) {
            // We transfer manager fee to `poolManager`
            uint256 poolManagerFee = Math.mulDiv(effectiveFee,
    poolManagerFeeBips, _FACTOR_ONE);
            if (poolManagerFee > 0) {
                token.safeTransfer(poolManager, poolManagerFee);
            }
        }
```

But when a user uses instant withdrawals in STEXAMM, the protocol charges `poolManagerFeeBips` for all swapped token1(all the output tokens) as `instantWithdrawalFee1`. This inflates the instant withdrawal fee and thus compromises the user.

Considering that `swapFee` is 1% and `poolManagerFee` is 20%, `poolManagerFee` should be charged is 0.2%, but the actual charge is 20%.

```
cache.instantWithdrawalFee1 = (amount1SwapEquivalent
    * ISovereignPool(pool).poolManagerFeeBips()) / BIPS;
amount1 += (amount1SwapEquivalent - cache.instantWithdrawalFee1);
```

Recommendations:

Note that this fix includes the fix for [M-3].

```
        if (_isInstantWithdrawal) {
            uint256 amount1SwapEquivalent = getAmountOut(token0, amount0);
            uint256 amount1WithFee = _withdrawalModule.convertToToken1(
                amount0);
            // Apply manager fee on instant withdrawals in token1

        cache.instantWithdrawalFee1 = (amount1SwapEquivalent * ISovereignPo
                ol(pool).poolManagerFeeBips()) / BIPS;
            amount1 += (amount1SwapEquivalent - cache.instantWithdrawalFee1);

        cache.instantWithdrawalFee1 = ((amount1WithFee - amount1SwapEquival
                ent) * ISovereignPool(pool).poolManagerFeeBips()) / BIPS;
            amount1 += amount1SwapEquivalent;

            amount0 = 0;
        }
```

Valantis: Resolved with @02591907ef...

Zenith: Verified

[H-2] Deposit and withdraw doesn't consider excess native tokens in withdrawalModule

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- STEXAMM.sol#L492-L497

Description:

When depositing, the protocol gets the total assets to calculate the shares minted to the user. All assets are converted to token1, the native token.

```
} else {
    (uint256 reserve0Pool, uint256 reserve1Pool)
    = ISovereignPool(pool).getReserves();
    // Account for token0 in pool (liquid) and pending unstaking (locked)
    uint256 reserve0Total = reserve0Pool
    + _withdrawalModule.amountToken0PendingUnstaking();
    // Account for token1 pending withdrawal to LPs (locked)
    uint256 reserve1PendingWithdrawal
    = _withdrawalModule.amountToken1PendingLPWithdrawal();
    // shares calculated in terms of token1
    shares = Math.mulDiv(
        _amount,
        totalSupplyCache,
        reserve1Pool + _withdrawalModule.amountToken1LendingPool()
            + _withdrawalModule.convertToToken1(reserve0Total)
    - reserve1PendingWithdrawal
    );
}
```

The total assets consist of the following parts:

1. all tokens1 in the pool (`reserve1Pool`)
2. tokens1 supplied to the LendingPool (`amountToken1LendingPool()`)
3. all tokens0 in the pool (`reserve0Pool`)

4. tokens0 that have been burned by the overseer but have not yet been claimed into the contract (`amountToken0PendingUnstaking`)
5. subtract the token1 that have not yet been claimed by the users (`reserve1PendingWithdrawal`)

```
reserve1Pool + _withdrawalModule.amountToken1LendingPool()
                    + _withdrawalModule.convertToToken1(reserve0Total)
    - reserve1PendingWithdrawal
```

Consider the following scenario, there are 110 token0 and 100 token1 in the pool.

1. Alice burns shares to claim 11 token0 and 10 token1, with 10 token0 queued up in the WithdrawalModule.
2. The owner calls `unstakeToken0Reserves()` to burn 20 token0 into 20 token1 for Alice and subsequent withdrawals.
3. The overseer then sends 20 token1 to the WithdrawalModule.
4. At this point, the total assets should be 110-20 = 90 token0 and 100 - 10 + (20 - 11) = 90 token1(in pool) + 9 token1(in WithdrawalModule) = 99 token1.
5. However, when Bob makes a deposit, reserve1Pool = 90 token1, amountToken1LendingPool = 0, reserve0Pool = 90, amountToken0PendingUnstaking = 0, reserve1PendingWithdrawal = 0, with total assets of 90 token0 and 90 token1, not taking into account the extra 9 token1 in the WithdrawalModule.

This is because in `WithdrawalModule.amountToken1PendingLPWithdrawal()`, if the available native tokens exceed _amountToken1PendingLPWithdrawal, these excess tokens will not be considered.

```
function amountToken1PendingLPWithdrawal()
    public view override returns (uint256) {
    uint256 excessNativeBalance = _getExcessNativeBalance();

    uint256 amountToken1PendingLPWithdrawalCache
    = _amountToken1PendingLPWithdrawal;
    if (amountToken1PendingLPWithdrawalCache > excessNativeBalance) {
        return amountToken1PendingLPWithdrawalCache - excessNativeBalance;
    } else {
        return 0;
    }
}
```

However, in `WithdrawalModule.update()`, these excess tokens will be sent back to the Pool and thus included in reserve1Pool.

```
uint256 amountToken1PendingLPWithdrawalCache
    = _amountToken1PendingLPWithdrawal;
if (excessNativeBalance > amountToken1PendingLPWithdrawalCache) {
    excessNativeBalance -= amountToken1PendingLPWithdrawalCache;
    amountToken1ClaimableLPWithdrawal
    += amountToken1PendingLPWithdrawalCache;
    cumulativeAmountToken1ClaimableLPWithdrawal
    += amountToken1PendingLPWithdrawalCache;
    _amountToken1PendingLPWithdrawal = 0;
} else {
    _amountToken1PendingLPWithdrawal -= excessNativeBalance;
    amountToken1ClaimableLPWithdrawal += excessNativeBalance;
    cumulativeAmountToken1ClaimableLPWithdrawal += excessNativeBalance;
    excessNativeBalance = 0;
    return;
}

// Wrap native token into token1 and re-deposit into the pool
address token1Address = ISTEXAMM(stex).token1();
IWETH9 token1 = IWETH9(token1Address);

token1.deposit{value: excessNativeBalance}();
// Pool reserves are measured as balances, hence we can replenish it with
    token1
// by transfering directly
token1.safeTransfer(pool, excessNativeBalance);
```

This issue is also existing in `withdraw()`.

```
{
    uint256 totalSupplyCache = totalSupply();
    // Account for token1 pending withdrawal to LPs (locked)
    uint256 reserve1PendingWithdrawal
    = _withdrawalModule.amountToken1PendingLPWithdrawal();
    // pro-rata share of token0 reserves in pool (liquid), token0 reserves
    pending in withdrawal queue (locked)
    // minus LP amount already pending withdrawal
    amount0 = Math.mulDiv(
        cache.reserve0Pool
    + _withdrawalModule.amountToken0PendingUnstaking()
            - _withdrawalModule.convertToToken0(reserve1PendingWithdrawal),
        _shares,
        totalSupplyCache
    );
```

```
    cache.amount1LendingPool =
        Math.mulDiv(_withdrawalModule.amountToken1LendingPool(), _shares,
    totalSupplyCache);
    // token1 amount calculated as pro-rata share of token1 reserves in the
    pool (liquid)
    // plus pro-rata share of token1 reserves earning yield in lending pool
    (liquid, assuming lending pool is working correctly)
    amount1 = cache.amount1LendingPool + Math.mulDiv(cache.reserve1Pool,
    _shares, totalSupplyCache);
}
```

POC

```
function test_poc() public {
    // Tests withdrawal where token0 is sent to unstake via withdrawal module

    vm.deal(address(this), 210 ether);
    weth.deposit{value: 100 ether}();

    address recipient = makeAddr("RECIPIENT");

    weth.approve(address(stex), type(uint256).max);
    uint256 shares = stex.deposit(100e18, 0, block.timestamp, recipient);

    token0.mint{value: 110e18}(address(pool));

    (uint256 reserve0, uint256 reserve1) = pool.getReserves();
    console.log(reserve0); // 110e18
    console.log(reserve1); // 100e18

    shares = stex.balanceOf(recipient);
    assertGt(shares, 0);

    vm.startPrank(recipient);

    (uint256 amount0, uint256 amount1) = stex.withdraw(shares/10, 0, 0,
    block.timestamp, recipient, false, false);
    assertEq(weth.balanceOf(recipient), amount1);
    assertEq(token0.balanceOf(recipient), 0);
    vm.stopPrank();

    withdrawalModule.unstakeToken0Reserves(20e18);
    // Mocks the processing of unstaking token0 by direct transfer of ETH

    vm.deal(address(withdrawalModule), 20e18);
    (reserve0, reserve1) = pool.getReserves();
```

```
    console.log(reserve0); // 90e18
    console.log(reserve1); // 90e18+100
    console.log(withdrawalModule.amountToken1LendingPool()); // 0
    console.log(withdrawalModule.amountToken0PendingUnstaking()); // 0
    console.log(withdrawalModule.amountToken1PendingLPWithdrawal()); // 0
    withdrawalModule.update();
    (reserve0, reserve1) = pool.getReserves();
    console.log(reserve0); // 90e18
    console.log(reserve1); // 99e18 + 210
    console.log(withdrawalModule.amountToken1LendingPool()); // 0
    console.log(withdrawalModule.amountToken0PendingUnstaking()); // 0
    console.log(withdrawalModule.amountToken1PendingLPWithdrawal()); // 0
}
```

Recommendations:

```
    function amountToken1PendingLPWithdrawal() public view override returns (
       uint256) {

    function amountToken1PendingLPWithdrawal() public view override returns (
       int256) {
         uint256 excessNativeBalance = _getExcessNativeBalance();

         uint256 amountToken1PendingLPWithdrawalCache
    = _amountToken1PendingLPWithdrawal;
       if (amountToken1PendingLPWithdrawalCache > excessNativeBalance) {

    return amountToken1PendingLPWithdrawalCache - excessNativeBalance;
            return int256(amountToken1PendingLPWithdrawalCache) - int256(
                excessNativeBalance);
       } else {
            return 0;
       }
...

    uint256 reserve1PendingWithdrawal = _withdrawalModule.amountToken1P
                endingLPWithdrawal();

    int256 reserve1PendingWithdrawal = _withdrawalModule.amountToken1Pe
                ndingLPWithdrawal();
```

```
            // shares calculated in terms of token1
            shares = Math.mulDiv(
                _amount,
                totalSupplyCache,
                reserve1Pool + _withdrawalModule.amountToken1LendingPool()
                    + _withdrawalModule.convertToToken1(reserve0Total)
    - reserve1PendingWithdrawal
            );
...
        {
            uint256 totalSupplyCache = totalSupply();
            // Account for token1 pending withdrawal to LPs (locked)

    uint256 reserve1PendingWithdrawal = _withdrawalModule.amountToken1P
                endingLPWithdrawal();

    int256 reserve1PendingWithdrawal = _withdrawalModule.amountToken1Pe
                ndingLPWithdrawal();
            // pro-rata share of token0 reserves in pool (liquid), token0
    reserves pending in withdrawal queue (locked)
            // minus LP amount already pending withdrawal
            amount0 = Math.mulDiv(
                cache.reserve0Pool
    + _withdrawalModule.amountToken0PendingUnstaking()

    - _withdrawalModule.convertToToken0(reserve1PendingWithdraw
                    al),

    - _withdrawalModule.convertToToken0(reserve1PendingWithdraw
                    al > 0: uint256(reserve1PendingWithdrawal) : 0),
                _shares,
                totalSupplyCache
            );

            cache.amount1LendingPool =
                Math.mulDiv(_withdrawalModule.amountToken1LendingPool(),
    _shares, totalSupplyCache);
            // token1 amount calculated as pro-rata share of token1 reserves
    in the pool (liquid)
            // plus pro-rata share of token1 reserves earning yield in
    lending pool (liquid, assuming lending pool is working correctly)

    amount1 = cache.amount1LendingPool + Math.mulDiv(cache.reserve1Pool
                , _shares, totalSupplyCache);
```

```
amount1 = cache.amount1LendingPool + Math.mulDiv(cache.reserve1Pool

+ reserve1PendingWithdrawal < 0: uint256(-1 * reserve1PendingW
          ithdrawal) : 0, _shares, totalSupplyCache);
    }
```

Note that when withdrawing, since the native tokens in the WithdrawalModule have not yet been sent to the Pool, the withdrawal may fail due to insufficient balance, in which case the user needs to actively call *`update()`* to send the native tokens to the Pool.

Valantis: Resolved with @45db3fb57a...

Zenith: Verified.

4.2 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] Instant withdrawal could fail if most of the `token1` are provided to the lending pool

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- STEXAMM.sol#L629

Description:

When a user calls `withdraw` and sets `_isInstantWithdrawal` to true, the calculated `amount0` will be swapped to `amount1SwapEquivalent`, which is then added to the previously calculated `amount1`. The contract will then attempt to withdraw `amount1Remaining` from the Sovereign Pool, assuming it has sufficient reserves to cover the request.

```
    function withdraw(
        uint256 _shares,
        uint256 _amount0Min,
        uint256 _amount1Min,
        uint256 _deadline,
        address _recipient,
        bool _unwrapToNativeToken,
        bool _isInstantWithdrawal
    ) external override nonReentrant returns (uint256 amount0,
    uint256 amount1) {
        // ...
        if (amount1 + cache.instantWithdrawalFee1 >
    cache.amount1LendingPool) {
            // Withdraw due token1 amount from pool into this module,
            // witholding any due pool manager fees, and send remaining
    amount to recipient,
            // also unwrapping into native token if necessary
            cache.amount1Remaining = amount1 + cache.instantWithdrawalFee1
    - cache.amount1LendingPool;
```

```
>>>         ISovereignPool(pool).withdrawLiquidity(0,
    cache.amount1Remaining, msg.sender, address(this), new bytes(0));

            if (cache.amount1Remaining > cache.instantWithdrawalFee1) {
                if (_unwrapToNativeToken) {
                    IWETH9(token1).withdraw(cache.amount1Remaining
    - cache.instantWithdrawalFee1);
                    Address.sendValue(payable(_recipient),
    cache.amount1Remaining - cache.instantWithdrawalFee1);
                } else {
                    ERC20(token1).safeTransfer(_recipient,
    cache.amount1Remaining - cache.instantWithdrawalFee1);
                }
            }
        }

        emit Withdraw(msg.sender, _recipient, amount0, amount1, _shares);
    }
```

If most of the `token1` is currently in the lending pool, the request will fail, even if the amount of `token1` in the lending pool is enough to cover `amount1Remaining`.

Recommendations:

Check if the Sovereign pool has enough reserves to cover the withdrawal. If not, try to withdraw from the lending pool.

Valantis: Resolved with @a8f8385f5e...

Zenith: Verified.

[M-2] Inaccurate `amount1SwapEquivalent` when an instant withdrawal is performed

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- STEXAMM.sol#L585

Description:

When `withdraw` is called and `_isInstantWithdrawal` is set to true, it means the user wants to instantly withdraw their token0. The amount of token1 they receive is calculated using `getAmountOut`, which reduces the received token1 by the calculated fee.

```
    function withdraw(
        uint256 _shares,
        uint256 _amount0Min,
        uint256 _amount1Min,
        uint256 _deadline,
        address _recipient,
        bool _unwrapToNativeToken,
        bool _isInstantWithdrawal
    ) external override nonReentrant returns (uint256 amount0,
    uint256 amount1) {
        // ...

        {
            uint256 totalSupplyCache = totalSupply();
            // Account for token1 pending withdrawal to LPs (locked)
            uint256 reserve1PendingWithdrawal
    = _withdrawalModule.amountToken1PendingLPWithdrawal();
            // pro-rata share of token0 reserves in pool (liquid), token0
    reserves pending in withdrawal queue (locked)
            // minus LP amount already pending withdrawal
>>>         amount0 = Math.mulDiv(
                cache.reserve0Pool
    + _withdrawalModule.amountToken0PendingUnstaking()
                    -
    _withdrawalModule.convertToToken0(reserve1PendingWithdrawal),
```

```
                _shares,
                totalSupplyCache
            );

            cache.amount1LendingPool =
                Math.mulDiv(_withdrawalModule.amountToken1LendingPool(),
    _shares, totalSupplyCache);
            // token1 amount calculated as pro-rata share of token1 reserves
    in the pool (liquid)
            // plus pro-rata share of token1 reserves earning yield in
    lending pool (liquid, assuming lending pool is working correctly)
            amount1 = cache.amount1LendingPool
    + Math.mulDiv(cache.reserve1Pool, _shares, totalSupplyCache);
        }

        // This is equivalent to an instant swap into token1 (with an extra
    fee in token1),
        // and withdraw the total amount in token1
        if (_isInstantWithdrawal) {
>>>         uint256 amount1SwapEquivalent = getAmountOut(token0, amount0);
            // Apply manager fee on instant withdrawals in token1
            cache.instantWithdrawalFee1 = (amount1SwapEquivalent
    * ISovereignPool(pool).poolManagerFeeBips()) / BIPS;
            amount1 += (amount1SwapEquivalent
    - cache.instantWithdrawalFee1);

            amount0 = 0;
        }
// ...
}
```

```
    function getAmountOut(address _tokenIn, uint256 _amountIn)
    public view returns (uint256 amountOut) {
        if ((_tokenIn ≠ token0 && _tokenIn ≠ token1) || _amountIn == 0) {
            return 0;
        }

        address swapFeeModule = ISovereignPool(pool).swapFeeModule();
>>>     SwapFeeModuleData memory swapFeeData
    = ISwapFeeModuleMinimalView(swapFeeModule).getSwapFeeInBips(
            _tokenIn, address(0), _amountIn, address(0), new bytes(0)
        );

        uint256 amountInWithoutFee = Math.mulDiv(_amountIn, BIPS, BIPS
    + swapFeeData.feeInBips);
        bool isZeroToOne = _tokenIn == token0;
```

```
        // token0 balances might not be 1:1 mapped to token1 balances,
        // hence we rely on the withdrawalModule to convert it (e.g., if
    token0 balances represent shares)
        amountOut = isZeroToOne
            ? _withdrawalModule.convertToToken1(amountInWithoutFee)
            : _withdrawalModule.convertToToken0(amountInWithoutFee);
    }
```

Inside `STEXRatioSwapFeeModule.getSwapFeeInBips`, the fee is calculated based on the reserve ratio. The `_amountIn` provided is added to the total reserve. However, in the context of `withdraw`, the `amount0` is still part of the total reserve. Therefore, adding `amount0` again to the total reserve in this case results in an inaccurate `ratioBps`.

```
    function getSwapFeeInBips(
        address _tokenIn,
        address, /*_tokenOut*/
        uint256 _amountIn,
        address, /*_user*/
        bytes memory /*_swapFeeModuleContext*/
    ) external view override returns (SwapFeeModuleData memory
    swapFeeModuleData) {
        ISovereignPool poolInterface = ISovereignPool(pool);
        ISTEXAMM stexInterface = ISTEXAMM(poolInterface.alm());
        // Fee is only applied on token0 → token1 swaps
        if (_tokenIn == poolInterface.token0()) {
            (uint256 reserve0, uint256 reserve1)
    = poolInterface.getReserves();
            IWithdrawalModule withdrawalModuleInterface
    = IWithdrawalModule(stexInterface.withdrawalModule());

            uint256 amount0PendingUnstaking
    = withdrawalModuleInterface.amountToken0PendingUnstaking();
            uint256 amountToken0PendingLPWithdrawal =
                withdrawalModuleInterface.convertToToken0

    (withdrawalModuleInterface.amountToken1PendingLPWithdrawal());

>>>         uint256 amount0Total =
reserve0 + amount0PendingUnstaking + _amountIn
    - amountToken0PendingLPWithdrawal;

            FeeParams memory feeParamsCache = feeParams;
            uint256 feeInBips;

            uint256 reserve1Total = reserve1
    + withdrawalModuleInterface.amountToken1LendingPool();
```

```
        if (reserve1Total == 0) {
            revert
STEXRatioSwapFeeModule__getSwapFeeInBips_ZeroReserveToken1();
        }

        uint256 ratioBips = (amount0Total * BIPS) / reserve1Total;

        // ...
    }
}
```

Recommendations:

Consider adding additional param inside `getAmountOut`, if it from `withdraw` request, pass 0 `amountIn`.

```
function getAmountOut(address _tokenIn, uint256 _amountIn) public view
    returns (uint256 amountOut) {
function getAmountOut(address _tokenIn, uint256 _amountIn, bool
    withdrawRequest) public view returns (uint256 amountOut) {
    if ((_tokenIn != token0 && _tokenIn != token1) || _amountIn == 0) {
        return 0;
    }

    address swapFeeModule = ISovereignPool(pool).swapFeeModule();
    SwapFeeModuleData memory swapFeeData
= ISwapFeeModuleMinimalView(swapFeeModule).getSwapFeeInBips(
        _tokenIn, address(0), _amountIn, address(0), new bytes(0)

_tokenIn, address(0), withdrawRequest ? 0 : _amountIn, address(0)
            , new bytes(0)
    );

    uint256 amountInWithoutFee = Math.mulDiv(_amountIn, BIPS, BIPS
+ swapFeeData.feeInBips);
    bool isZeroToOne = _tokenIn == token0;
    // token0 balances might not be 1:1 mapped to token1 balances,
    // hence we rely on the withdrawalModule to convert it (e.g., if
token0 balances represent shares)
    amountOut = isZeroToOne
        ? _withdrawalModule.convertToToken1(amountInWithoutFee)
        : _withdrawalModule.convertToToken0(amountInWithoutFee);
```

```
    }
```

Valantis: Resolved with @6425609f446...

Zenith: Verified.

[M-3] Overcharging of instant withdrawal fees

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- STEXAMM.sol#L584-L591

Description:

The instant withdrawal fee models the swap of token0 to token1, so it should charge the fee similar to `SovereignPool.swap()`.

But the problems is, `getAmountOut()` has already charged the fee once, it reduces the output token1, but then `instantWithdrawalFee1` has charged again. For example, consider stHype:Hype = 1:1, and the swap fee is 1%. When the user withdraws 1000 stHype instantly, `getAmountOut()` returns 990.1 Hype. At this time, the swap fee has been charged once, but the protocol will charge another `instantWithdrawalFee1` of `990.1 * poolManagerFeeBips / BIPS`.

```
if (_isInstantWithdrawal) {
    uint256 amount1SwapEquivalent = getAmountOut(token0, amount0);
    // Apply manager fee on instant withdrawals in token1
    cache.instantWithdrawalFee1 = (amount1SwapEquivalent
    * ISovereignPool(pool).poolManagerFeeBips()) / BIPS;
    amount1 += (amount1SwapEquivalent - cache.instantWithdrawalFee1);

    amount0 = 0;
}
```

This is because `instantWithdrawalFee1` has already been charged in `getAmountOut()` and should not be charged again to the user.

Recommendations:

Note that this fix includes the fix for [H-1].

```
        if (_isInstantWithdrawal) {
            uint256 amount1SwapEquivalent = getAmountOut(token0, amount0);
```

```
            uint256 amount1WithFee = _withdrawalModule.convertToToken1(
                amount0);
            // Apply manager fee on instant withdrawals in token1

        cache.instantWithdrawalFee1 = (amount1SwapEquivalent * ISovereignPo
                ol(pool).poolManagerFeeBips()) / BIPS;
            amount1 += (amount1SwapEquivalent - cache.instantWithdrawalFee1);

        cache.instantWithdrawalFee1 = ((amount1WithFee - amount1SwapEquival
                ent) * ISovereignPool(pool).poolManagerFeeBips()) / BIPS;
            amount1 += amount1SwapEquivalent;

            amount0 = 0;
        }
```

Valantis: Resolved with @02591907ef...

Zenith: Verified.

4.3 Low Risk

A total of 1 low risk findings were identified.

[L-1] `claim` should trigger `update` to prevent reverts

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- stHYPEWithdrawalModule.sol#L522-L552

Description:

`claim` allows the caller to redeem an `LPWithdrawals` request, which depends on `amountToken1ClaimableLPWithdrawal` and `cumulativeAmountToken1ClaimableLPWithdrawal`. These values are calculated and updated when `update` is triggered.

If a user calls `claim` without first triggering `update`, the request may revert due to not using the latest values of `amountToken1ClaimableLPWithdrawal` and `cumulativeAmountToken1ClaimableLPWithdrawal`.

Recommendations:

Trigger `update` logic before `claim` is executed.

Valantis: Acknowledged. Our keeper frequently checks if update can be called, and in our Deposit UI we simulate claims before letting users execute the transaction.