Jan.24

SECURITY REVIEW REPORT FOR VALANTIS

CONTENTS

- About Hexens / 4
- Audit led by / 5
- Methodology / 6
- Severity structure / 7
- Executive summary / 9
- Scope / 10
- Summary / 11
- Weaknesses / 12
  - Missing deadline parameter in swap / 12
  - Renouncing pool manager won't claim fees in case of external sovereign vault / 17
  - Gas optimisation of deposit functionality / 19
  - Centralisation risk in swap fee / 22
  - Constant variables should be marked as private / 23
  - Unused errors / 25
  - Storage loading gas optimisations / 26
  - Magic numbers should be replaced with constants / 27
  - Sovereign Pool Factory CREATE2 nonce missing chain ID / 28
  - Order of checks gas optimisation / 29

CONTENTS

- Gas optimisation of double zero check / 31
- Missing events on pool manager fee claiming / 33
- Unused events / 35
- Unused imports / 36

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date	Audit Completion Date
05.01.2024	15.01.2024

---

hexens × VALANTIS

METHODOLOGY

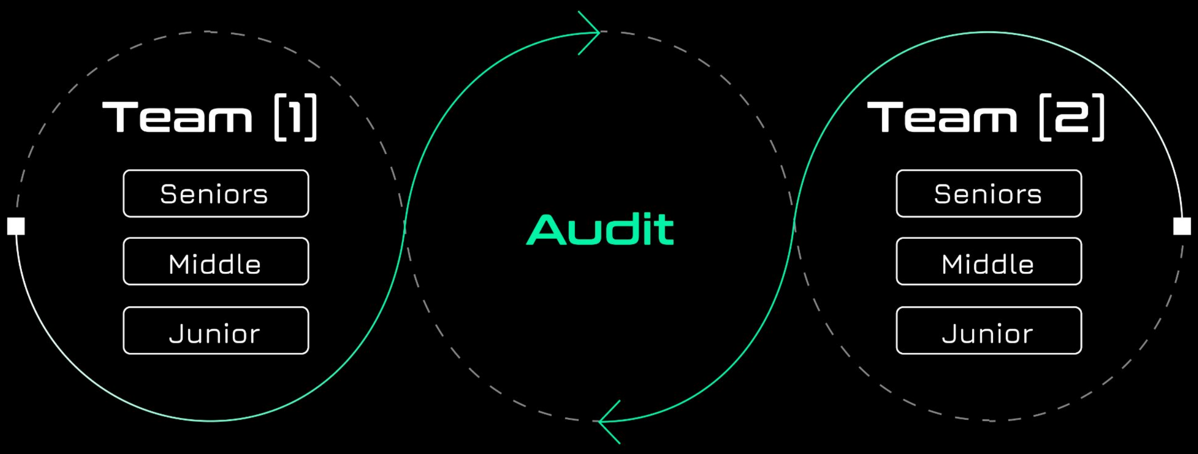
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor*
Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered the Sovereign Pool contract and its imported dependencies of the Valantis protocol. Valantis is a next-generation DEX, built with a focus on modularity and user experience.

Our security assessment was a full review of the smart contracts spanning a total of 1 week.

During our audit, we have identified 2 medium severity vulnerabilities. The swapping mechanism did not protect users with a deadline and there was an edge case where pool manager fees would be discarded.

We have also identified various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/ValantisLabs/valantis-core/tree/89e1ea346ec3f324e943d04222573c1683463c99

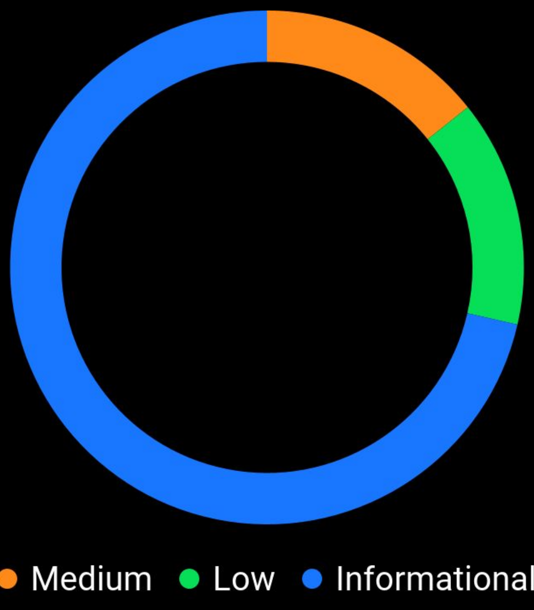
The issues described in this report were fixed in the following commit:
https://github.com/ValantisLabs/valantis-core/commit/9207b80a5aacf7b4ac0c01a66b54092216f447ed

SUMMARY

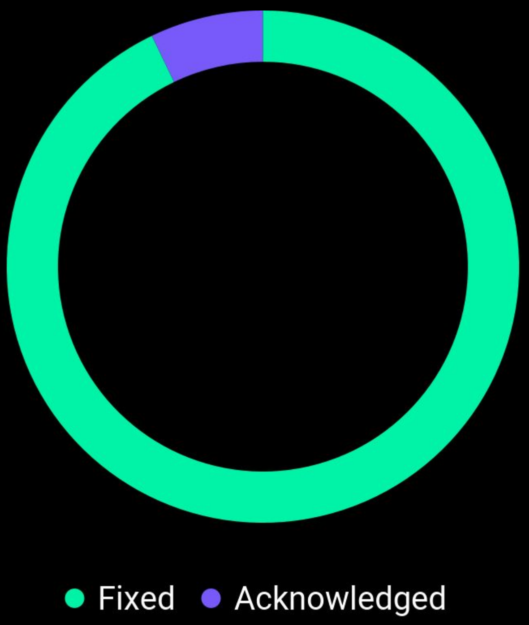
SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	0
MEDIUM	2
LOW	2
INFORMATIONAL	10

TOTAL: 14

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

VLTS-4. MISSING DEADLINE PARAMETER IN SWAP

SEVERITY: Medium

PATH: SovereignPool.sol:swap:L676-819

REMEDIATION: allow users to set a fixed deadline as a parameter and never automatically assign the deadline as block.timestamp within the function, as that would effectively mean that the transaction has no deadline

STATUS: fixed

DESCRIPTION:

Although the **SovereignPool.sol** contract's **swap** function incorporates slippage protection, it lacks a crucial element: the **deadline**. This absence leaves transactions without a specified deadline, potentially allowing them to be included at any time by validators while they are pending in the mempool.

This means that it might result in transactions lingering in the mempool and being executed much later than intended by the user. Consequently, users or the protocol might receive unfavorable prices, as validators could delay transaction inclusion to manipulate trade outcomes.

```
function swap(
    SovereignPoolSwapParams calldata _swapParams
) external override nonReentrant returns (uint256 amountInUsed, uint256 amountOut) {
    // Cannot swap below minimum input token amount
    if (_swapParams.amountIn == 0) {
        revert SovereignPool__swap_insufficientAmountIn();
    }

    if (_swapParams.recipient == address(0)) {
        revert SovereignPool__swap_invalidRecipient();
    }

    SwapCache memory swapCache = SwapCache({
        swapFeeModule: _swapFeeModule,
        tokenInPool: _swapParams.isZeroToOne ? _token0 : _token1,
        tokenOutPool: _swapParams.isZeroToOne ? _token1 : _token0,
        amountInWithoutFee: 0
    });

    if (_swapParams.swapTokenOut == address(0) || _swapParams.swapTokenOut == address(swapCache.tokenInPool))
{
        revert SovereignPool__swap_invalidSwapTokenOut();
    }

    // If reserves are kept in the pool, only token0 <-> token1 swaps are allowed
    if (sovereignVault == address(this) && _swapParams.swapTokenOut != address(swapCache.tokenOutPool)) {
        revert SovereignPool__swap_invalidPoolTokenOut();
    }

    bytes memory verifierData;
    if (address(_verifierModule) != address(0)) {
        // Query Verifier Module to authenticate the swap
        verifierData = _verifyPermission(
            msg.sender,
            _swapParams.swapContext.verifierContext,
            uint8(AccessType.SWAP)
        );
    }
    // Calculate swap fee in bips
    SwapFeeModuleData memory swapFeeModuleData = address(swapCache.swapFeeModule) != address(0)
      ? swapCache.swapFeeModule.getSwapFeeInBips(
          _swapParams.isZeroToOne,
          _swapParams.amountIn,
          msg.sender,
          _swapParams.swapContext.swapFeeModuleContext
      )
      : SwapFeeModuleData({ feeInBips: defaultSwapFeeBips, internalContext: new bytes(0) });

    if (swapFeeModuleData.feeInBips > MAX_SWAP_FEE_BIPS) {
      revert SovereignPool__swap_excessiveSwapFee();
    }

    // Since we do not yet know how much of `amountIn` will be filled,
    // this quantity is calculated in such a way that `msg.sender`
    // will be charged `feeInBips` of whatever the amount of tokenIn filled
    // ends up being (see docs for more details)
    swapCache.amountInWithoutFee = Math.mulDiv(_swapParams.amountIn, 1e4, 1e4 +
swapFeeModuleData.feeInBips); // @audit-issue wrong calculation

    ALMLiquidityQuote memory liquidityQuote = ISovereignALM(alm).getLiquidityQuote(
      ALMLiquidityQuoteInput({
          isZeroToOne: _swapParams.isZeroToOne,
          amountInMinusFee: swapCache.amountInWithoutFee,
          fee: _swapParams.amountIn - swapCache.amountInWithoutFee,
          sender: msg.sender,
          recipient: _swapParams.recipient,
          tokenOutSwap: _swapParams.swapTokenOut
      }),
      _swapParams.swapContext.externalContext,
      verifierData
    );

    amountOut = liquidityQuote.amountOut;
```

```
if (
    !_checkLiquidityQuote(
        _swapParams.isZeroToOne,
        swapCache.amountInWithoutFee,
        liquidityQuote.amountInFilled,
        amountOut,
        _swapParams.amountOutMin
    )
) {
    revert SovereignPool__swap_invalidLiquidityQuote();
}

// If amountOut is 0, we do not transfer any input token
if (amountOut == 0) {
    return (0, 0);
}

// Calculate the actual swap fee to be charged in input token (`effectiveFee`),
// now that we know the tokenIn amount filled
uint256 effectiveFee;
if (liquidityQuote.amountInFilled != swapCache.amountInWithoutFee) {
    effectiveFee = Math.mulDiv(liquidityQuote.amountInFilled, swapFeeModuleData.feeInBips, 1e4);
    amountInUsed = liquidityQuote.amountInFilled + effectiveFee;
} else {
    // Using above formula in case amountInWithoutFee == amountInFilled introduces rounding errors
    effectiveFee = _swapParams.amountIn - swapCache.amountInWithoutFee;
    amountInUsed = _swapParams.amountIn;
}

_handleTokenInTransfersOnSwap(
    _swapParams.isZeroToOne,
    _swapParams.isSwapCallback,
    swapCache.tokenInPool,
    amountInUsed,
    effectiveFee,
    _swapParams.swapContext.swapCallbackContext
);
```

```
 // Update internal state and oracle module.
      // In case of rebase tokens, `amountInUsed` and `amountOut` might not match
      // the exact balance deltas due to rounding errors.
      _updatePoolStateOnSwap(_swapParams.isZeroToOne, amountInUsed, amountOut, effectiveFee);

      if (
         address(_sovereignOracleModule) != address(0) &&
         _swapParams.swapTokenOut == address(swapCache.tokenOutPool) &&
         amountInUsed > 0 &&
         amountOut > 0
      ) {
         _sovereignOracleModule.writeOracleUpdate(_swapParams.isZeroToOne, amountInUsed, effectiveFee,
amountOut);
      }

      // Transfer `amountOut to recipient
      _handleTokenOutTransferOnSwap(IERC20(_swapParams.swapTokenOut), _swapParams.recipient, amountOut);

      // Update state for Swap fee module,
      // only performed if internalContext is non-empty
      if (
         address(swapCache.swapFeeModule) != address(0) &&
         keccak256(swapFeeModuleData.internalContext) != keccak256(new bytes(0))
      ) {
         swapCache.swapFeeModule.callbackOnSwapEnd(effectiveFee, amountInUsed, amountOut,
swapFeeModuleData);
      }

      // Perform post-swap callback to liquidity module if necessary
      if (liquidityQuote.isCallbackOnSwap) {
         ISovereignALM(alm).onSwapCallback(_swapParams.isZeroToOne, amountInUsed, amountOut);
      }

      emit Swap(msg.sender, _swapParams.isZeroToOne, amountInUsed, effectiveFee, amountOut);
   }
```

VLTS-14. RENOUNCING POOL MANAGER WON'T CLAIM FEES IN CASE OF EXTERNAL SOVEREIGN VAULT

SEVERITY: Medium

PATH: SovereignPool.sol:setPoolManager:L439-455

REMEDIATION: we would recommend to also handle outstanding pool manager fees when the pool manager renounces the role for a pool with an external vault

STATUS: fixed

DESCRIPTION:

The pool manager of a sovereign pool is able to claim the outstanding pool manager fees through the **claimPoolManagerFees** function.

If the pool does not have a sovereign vault, then the fees are taken from the local storage variables **feePoolManager0** and **feePoolManager1** and transferred to the manager. If it does have an external sovereign vault, then the fees are claimed from the sovereign vault, by calling **SovereignVault:claimPoolManagerFees** and then transferred to the manager.

When **setPoolManager(address(0))** is called, it effectively renounces the role of pool manager. In the first case without a sovereign vault, the outstanding pool manager fees are directly paid to the pool manager before setting the fee to 0.

However, in the latter case with an external sovereign vault, the outstanding fees are not paid out and the fees and corresponding reserved assets would be stuck in the contract.

If the pool manager had first called **claimPoolManagerFees** before renouncing the role with **setPoolManager**, then the fees would've been paid.

This is an unnecessary discrepancy between the two configurations and will lead to confusion and potentially stuck assets.

```
function setPoolManager(address _manager) external override onlyPoolManager nonReentrant {
  poolManager = _manager;

  if (_manager == address(0)) {
     poolManagerFeeBips = 0;
     // It will be assumed pool is not going to contribute anything to protocol fees.
     if (sovereignVault == address(this)) {
        if (feePoolManager0 > 0) _token0.safeTransfer(msg.sender, feePoolManager0);
        if (feePoolManager1 > 0) _token1.safeTransfer(msg.sender, feePoolManager1);

        feePoolManager0 = 0;
        feePoolManager1 = 0;
     }
  }

  emit PoolManagerSet(_manager);
}
```

VLTS-7. GAS OPTIMISATION OF DEPOSIT FUNCTIONALITY

SEVERITY: Low

PATH: SovereignPool.sol:depositLiquidity:L831-889

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The **depositLiquidity** function currently conducts checks for both **token0** and **token1** even when the deposit amount for one of the tokens is 0. This can be observed in scenarios where the deposit amount e.g. for **token0** is **0**, yet the function processes checks related to **token0**.

```
// Post-deposit checks for token0
if (isToken0Rebase) {
    uint256 amount0AbsDiff = amount0Deposited < _amount0
        ? _amount0 - amount0Deposited
        : amount0Deposited - _amount0;

    if (amount0AbsDiff > token0AbsErrorTolerance) {
        revert SovereignPool__depositLiquidity_excessiveToken0ErrorOnTransfer();
    }
} else {
    if (amount0Deposited != _amount0) revert SovereignPool__depositLiquidity_insufficientToken0Amount();

    if (amount0Deposited > 0) _reserve0 += amount0Deposited;
}
```

This approach results in unnecessary gas consumption due to the execution of checks and validations for a token when the corresponding deposit amount is zero.

```
function depositLiquidity(
    uint256 _amount0,
    uint256 _amount1,
    address _sender,
    bytes calldata _verificationContext,
    bytes calldata _depositData
) external override onlyALM nonReentrant returns (uint256 amount0Deposited, uint256 amount1Deposited) {
    // We disable deposits,
    // since reserves are not meant to be stored in the pool
    if (sovereignVault != address(this)) revert SovereignPool__depositLiquidity_depositDisabled();

    uint256 token0PreBalance = _token0.balanceOf(address(this));
    uint256 token1PreBalance = _token1.balanceOf(address(this));

    if (_amount0 == 0 && _amount1 == 0) {
        revert SovereignPool__depositLiquidity_zeroTotalDepositAmount();
    }

    if (address(_verifierModule) != address(0)) {
        _verifyPermission(_sender, _verificationContext, uint8(AccessType.DEPOSIT));
    }

    ISovereignALM(msg.sender).onDepositLiquidityCallback(_amount0, _amount1, _depositData);

    amount0Deposited = _token0.balanceOf(address(this)) - token0PreBalance;
    amount1Deposited = _token1.balanceOf(address(this)) - token1PreBalance;

    // Post-deposit checks for token0
    if (isToken0Rebase) {
        uint256 amount0AbsDiff = amount0Deposited < _amount0
            ? _amount0 - amount0Deposited
            : amount0Deposited - _amount0;

        if (amount0AbsDiff > token0AbsErrorTolerance) {
            revert SovereignPool__depositLiquidity_excessiveToken0ErrorOnTransfer();
        }
    } else {
        if (amount0Deposited != _amount0) revert SovereignPool__depositLiquidity_insufficientToken0Amount();

        if (amount0Deposited > 0) _reserve0 += amount0Deposited;
    }
    // Post-deposit checks for token1
    if (isToken1Rebase) {
      uint256 amount1AbsDiff = amount1Deposited < _amount1
        ? _amount1 - amount1Deposited
        : amount1Deposited - _amount1;

      if (amount1AbsDiff > token1AbsErrorTolerance) {
        revert SovereignPool__depositLiquidity_excessiveToken1ErrorOnTransfer();
      }
    } else {
      if (amount1Deposited != _amount1) revert SovereignPool__depositLiquidity_insufficientToken1Amount();

      if (amount1Deposited > 0) _reserve1 += amount1Deposited;
    }

    emit DepositLiquidity(amount0Deposited, amount1Deposited);
  }
```

Revise the **depositLiquidity** function to conditionally skip checks for a token when the deposit amount for that token (**_amount0** / **_amount1**) is 0.

An example for token0:

```
        // Post-deposit checks for token0
+       if (_amount0 != 0) {
          if (isToken0Rebase) {
            uint256 amount0AbsDiff = amount0Deposited < _amount0
              ? _amount0 - amount0Deposited
              : amount0Deposited - _amount0;

            if (amount0AbsDiff > token0AbsErrorTolerance) {
              revert SovereignPool__depositLiquidity_excessiveToken0ErrorOnTransfer();
            }
          } else {
            if (amount0Deposited != _amount0) revert SovereignPool__depositLiquidity_insufficientToken0Amount();

            if (amount0Deposited > 0) _reserve0 += amount0Deposited;
          }
+       }
```

VLTS-8. CENTRALISATION RISK IN SWAP FEE

SEVERITY: Low

PATH: SovereignPool.sol:setSwapFeeModule:L513-517

REMEDIATION: set a time delay before allowing the pool manager to swap the fee module

STATUS: fixed

DESCRIPTION:

The swap fee module can be changed without any delays via the **setSwapFeeModule** function on line 513. This enables a pool manager to front run a user swap and increase the fee, leading to user unexpectedly paying more fees.

The **minAmountOut** slippage protection does protect the user against this and the pool manager would only be able to maximise the fee against this amount.

However, if this is set to zero, then the increase in fee can be quite steep given the max fee is 50%.

```
function setSwapFeeModule(address swapFeeModule_) external override onlyPoolManager nonReentrant {
    _swapFeeModule = ISwapFeeModule(swapFeeModule_);

    emit SwapFeeModuleSet(swapFeeModule_);
}
```

VLTS-2. CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH: SovereignPool.sol

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The MAX_SWAP_FEE_BIPS and MAX_POOL_MANAGER_FEE_BIPS parameters on lines 101 and 106 should be **private**. Setting constants to private will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the values outside of where it's used, and won't add another entry to the method ID table. The values can still be read from the verified contract source code if necessary.

```
/**
   @notice Maximum swap fee is 50% of input amount.
   @dev See docs for a more detailed explanation about how swap fees are applied.
 */
uint256 public constant MAX_SWAP_FEE_BIPS = 10_000; // @audit private

/**
   @notice `poolManager` can collect up to 50% of swap fees.
 */
uint256 public constant MAX_POOL_MANAGER_FEE_BIPS = 5_000;
```

```
--    uint256 public constant MAX_SWAP_FEE_BIPS = 10_000;
--    uint256 public constant MAX_POOL_MANAGER_FEE_BIPS = 5_000;
++    uint256 private constant MAX_SWAP_FEE_BIPS = 10_000;
++    uint256 private constant MAX_POOL_MANAGER_FEE_BIPS = 5_000;
```

VLTS-5. UNUSED ERRORS

SEVERITY: Informational

PATH: SovereignPool.sol:L84

REMEDIATION: if the error is redundant and there is no missing logic where it can be used, it should be removed

STATUS: fixed

DESCRIPTION:

The SovereignPool contract declares the **SovereignPool__swap_insufficientRecipientAmountOut** error but it is never used.

```
error SovereignPool__swap_insufficientRecipientAmountOut();
```

VLTS-9. STORAGE LOADING GAS OPTIMISATIONS

SEVERITY: Informational

PATH: SovereignPool.sol:claimPoolManagerFees:L581-639

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

On line 603 two storage variables are used as input for **claimPoolManagerFees**. However, such variables have already been previously loaded unto the stack on line 959 via **getPoolManagerFees**. Therefore, gas could be saved by using the values stored instead of calling the storage variable directly.

```
(feePoolManager0Received, feePoolManager1Received) = getPoolManagerFees();

// Attempt to claim pool manager fees from `sovereignVault`
// This is necessary since in this case reserves are not kept in this pool
if (sovereignVault != address(this)) {
    uint256 token0PreBalance = _token0.balanceOf(address(this));
    uint256 token1PreBalance = _token1.balanceOf(address(this));

    ISovereignVaultMinimal(sovereignVault).claimPoolManagerFees(feePoolManager0, feePoolManager1);
```

```
- ISovereignVaultMinimal(sovereignVault).claimPoolManagerFees(feePoolManager0, feePoolManager1);
+ ISovereignVaultMinimal(sovereignVault).claimPoolManagerFees(feePoolManager0Received,
feePoolManager1Received);
```

VLTS-10. MAGIC NUMBERS SHOULD BE REPLACED WITH CONSTANTS

SEVERITY: Informational

PATH: SovereignPool.sol

REMEDIATION: change hard coded values such as 1e4 with a constant that explains the purpose of such variable, for example as MAX_SWAP_FEE_BIPS

STATUS: fixed

DESCRIPTION:

Magic numbers, e.g. **1e4** hurt readability and use more gas. Replacing them with a constant like **MAX_SWAP_FEE_BIPS** improves readability and gas cost.

```
if (_feeProtocol0Bips > 1e4 || _feeProtocol1Bips > 1e4) {
        revert SovereignPool__claimPoolManagerFees_invalidProtocolFee();
    }
```

```
uint256 protocolFee0 = Math.mulDiv(_feeProtocol0Bips, feePoolManager0Received, 1e4);uint256 protocolFee1 =
Math.mulDiv(_feeProtocol1Bips, feePoolManager1Received, 1e4);
```

```
swapCache.amountInWithoutFee = Math.mulDiv(_swapParams.amountIn, 1e4, 1e4 + swapFeeModuleData.feeInBips);
```

```
effectiveFee = Math.mulDiv(liquidityQuote.amountInFilled, swapFeeModuleData.feeInBips, 1e4);
```

```
uint256 poolManagerFee = Math.mulDiv(effectiveFee, poolManagerFeeBips, 1e4);
```

VLTS-11. SOVEREIGN POOL FACTORY CREATE2 NONCE MISSING CHAIN ID

SEVERITY: Informational

PATH: SovereignPoolFactory.sol:deploy:L23-32

REMEDIATION: if unique SovereignPool addresses are desired, then the deployment function should be made chain-specific using the chain.id in the nonce

STATUS: fixed

DESCRIPTION:

The deployment function for a SovereignPool uses the CREATE2 opcode with the hash of an incrementing nonce as salt. Because CREATE2 also uses the creating contract's address, the calculated address will be unique to the chain.

However, in the case that the SovereignPoolFactory has been deployed to the same address on another chain, then it will start again from a nonce of zero with the same address and deploy to the same addresses.

```
function deploy(bytes32, bytes calldata _constructorArgs) external override returns (address deployment) {
    SovereignPoolConstructorArgs memory args = abi.decode(_constructorArgs, (SovereignPoolConstructorArgs));

    // Salt to trigger a create2 deployment,
    // as create is prone to re-org attacks
    bytes32 salt = keccak256(abi.encode(nonce));
    deployment = address(new SovereignPool{ salt: salt }(args));

    nonce++;
}
```

VLTS-12. ORDER OF CHECKS GAS OPTIMISATION

SEVERITY: Informational

PATH: SovereignPool.sol:depositLiquidity:L831-889

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In the **depositLiquidity** function, both **_amount0** and **_amount1** parameters are checked against zero on line 845. This happens after the token 0 and token 1 balances are fetched.

Since the parameter checks are invariant of the prior static function calls, the code could be gas optimised for reverting operations if the checks are done before fetching the balances.

```
function depositLiquidity(
  uint256 _amount0,
  uint256 _amount1,
  address _sender,
  bytes calldata _verificationContext,
  bytes calldata _depositData
) external override onlyALM nonReentrant returns (uint256 amount0Deposited, uint256 amount1Deposited) {
  // We disable deposits,
  // since reserves are not meant to be stored in the pool
  if (sovereignVault != address(this)) revert SovereignPool__depositLiquidity_depositDisabled();

  uint256 token0PreBalance = _token0.balanceOf(address(this));
  uint256 token1PreBalance = _token1.balanceOf(address(this));

  if (_amount0 == 0 && _amount1 == 0) {
     revert SovereignPool__depositLiquidity_zeroTotalDepositAmount();
  }

  [..]
}
```

We recommend to switch the balanceOf calls and the parameter checks:

```
  function deploy(bytes32, bytes calldata _constructorArgs) external override returns (address deployment) {
     SovereignPoolConstructorArgs memory args = abi.decode(_constructorArgs, (SovereignPoolConstructorArgs));

     // Salt to trigger a create2 deployment,
     // as create is prone to re-org attacks
     bytes32 salt = keccak256(abi.encode(nonce));
     deployment = address(new SovereignPool{ salt: salt }(args));

     nonce++;
  }
```

VLTS-13. GAS OPTIMISATION OF DOUBLE ZERO CHECK

SEVERITY: Informational

PATH: SovereignPool.sol:depositLiquidity:L831-889

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In the **depositLiquidity** function, the parameters **_amount0** and **_amount1** are both checked against zero and an error is thrown if both are zero.

This check can be gas optimised by making use of one check on the final value from an **OR** operation on the concatenation of all integers.

```
function depositLiquidity(
  uint256 _amount0,
  uint256 _amount1,
  address _sender,
  bytes calldata _verificationContext,
  bytes calldata _depositData
) external override onlyALM nonReentrant returns (uint256 amount0Deposited, uint256 amount1Deposited) {
  // We disable deposits,
  // since reserves are not meant to be stored in the pool
  if (sovereignVault != address(this)) revert SovereignPool__depositLiquidity_depositDisabled();

  uint256 token0PreBalance = _token0.balanceOf(address(this));
  uint256 token1PreBalance = _token1.balanceOf(address(this));

  if (_amount0 == 0 && _amount1 == 0) {
     revert SovereignPool__depositLiquidity_zeroTotalDepositAmount();
  }

  [..]
}
```

Replace the multiple zero checks with one check and OR operation.

For example:

```
if (_amount0 | _amount1 == 0) {
   revert SovereignPool__depositLiquidity_zeroTotalDepositAmount();
}
```

VLTS-15. MISSING EVENTS ON POOL MANAGER FEE CLAIMING

SEVERITY: Informational

PATH: SovereignPool.sol:setPoolManager, _handleTokenInTransfersOnSwap:L439-455, L958-1001

REMEDIATION: emit the PoolManagerFeesClaimed in all cases where some pool manager fees are sent to the pool manager

STATUS: acknowledged

DESCRIPTION:

The function **setPoolManager** will automatically claim any outstanding pool manager fees for the pool manager if the new pool manager is to be set to **address(0)**. The function **_handleTokenInTransfersOnSwap** will immediately send the pool manager fees on the amount from a swap to the pool manager if the token is a rebaseable token.

However, neither function emits the **PoolManagerFeesClaimed** event, which is emitted in the **claimPoolManagerFees** function.

This makes off-chain tracking of the fees more difficult and incomplete if done using the event only.

```
function setPoolManager(address _manager) external override onlyPoolManager nonReentrant {
   poolManager = _manager;

   if (_manager == address(0)) {
      poolManagerFeeBips = 0;
      // It will be assumed pool is not going to contribute anything to protocol fees.
      if (sovereignVault == address(this)) {
         if (feePoolManager0 > 0) _token0.safeTransfer(msg.sender, feePoolManager0);
         if (feePoolManager1 > 0) _token1.safeTransfer(msg.sender, feePoolManager1);

         feePoolManager0 = 0;
         feePoolManager1 = 0;
      }
   }

   emit PoolManagerSet(_manager);
}

function _handleTokenInTransfersOnSwap(
   bool isZeroToOne,
   bool isSwapCallback,
   IERC20 token,
   uint256 amountInUsed,
   uint256 effectiveFee,
   bytes calldata _swapCallbackContext
) private {
   [..]
   if (isTokenInRebase && sovereignVault == address(this) && poolManager != address(0)) {
      // We transfer manager fee to `poolManager`
      uint256 poolManagerFee = Math.mulDiv(effectiveFee, poolManagerFeeBips, 1e4);
      if (poolManagerFee > 0) {
         token.safeTransfer(poolManager, poolManagerFee);
      }
   }
}
```

VLTS-16. UNUSED EVENTS

SEVERITY: Informational

PATH: ISovereignPool.sol:L12

REMEDIATION: if events serve no purpose in the current or future versions, remove them to declutter the code

STATUS: fixed

DESCRIPTION:

The ISovereignPool interface declares the VerifierModuleSet event but it is never used.

```
event VerifierModuleSet(address verifierModule);
```

VLTS-17. UNUSED IMPORTS

SEVERITY: Informational

PATH: SovereignPool.sol:L20

REMEDIATION: If structs serve no purpose in the current or future versions, remove them to declutter the code.

STATUS: fixed

DESCRIPTION:

The SovereignPool contract imports the SovereignPoolSwapContextData struct but it is never used.

```
import {
  SovereignPoolConstructorArgs,
  SovereignPoolSwapContextData,
  SwapCache,
  SovereignPoolSwapParams
} from 'src/pools/structs/SovereignPoolStructs.sol';
```