hexens × Valantis

MAR.25

SECURITY REVIEW REPORT FOR VALANTIS

CONTENTS

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.

EXECUTIVE SUMMARY

OVERVIEW

This report covers the results of our engagement with Valantis to audit their Stake Exchange AMM (STEX AMM) and the first implemented module for stHYPE.

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

During our audit, we have identified two critical severity vulnerabilities, both of which could have resulted in direct theft of user principal assets.

We have also identified several minor severity vulnerabilities and code optimisations.

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-stex/tree/25a19b663f86b53112a5e020c843904a571cc1e8

The issues described in this report were fixed in the following commit:

https://github.com/ValantisLabs/valantis-stex/tree/95122c7693f9516385aef330ef36bb1ccec2cb94

AUDITING DETAILS

STARTED
10.03.2025

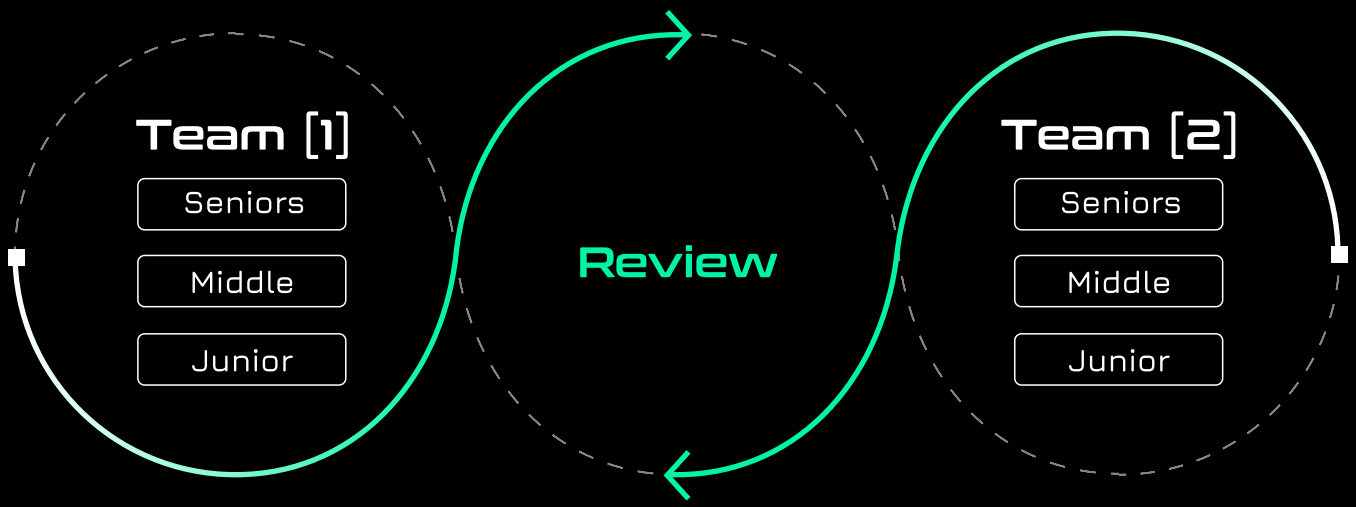
DELIVERED
17.03.2025

Review Led by

KASPER ZWIJSEN
Head of Audits | Hexens

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

Smart contract 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 smart contract 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.

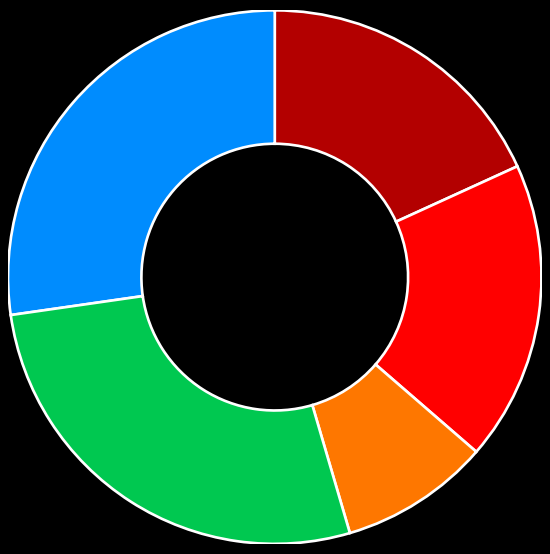
ISSUE SYMBOLIC CODES

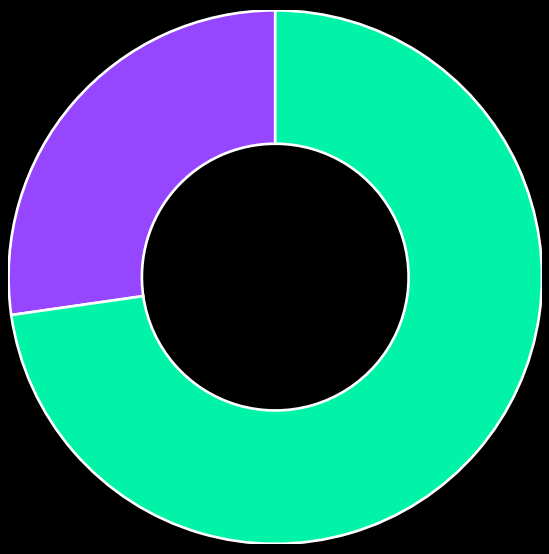
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	2
High	2
Medium	1
Low	3
Informational	3

Total: 11





WEAKNESSES

This section contains the list of discovered weaknesses.

VLTS3-5

THE CALCULATION OF SHARES IN STEXAMM IS INCORRECT BECAUSE OF USING THE OUTDATED AMOUNTTOKEN1PENDINGLPWITHDRAWAL

SEVERITY: Critical

PATH:

src/STEXAMM.sol#L468-L479

src/STEXAMM.sol#L550-L555

REMEDIATION:

You should trigger update() whenever a deposit or withdrawal occurs, or create a getter function for amountToken1PendingLPWithdrawal to keep it updated at all times.

STATUS: Fixed

DESCRIPTION:

The calculation of shares in **deposit()** function of STEXAMM contract:

```
(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
);
```

amountToken0PendingUnstaking() is the amount that the withdrawal module is waiting for the overseer contract to claim after **unstakeToken0Reserves**. In **stHYPEWithdrawalModule**, this getter function is always up to date:

```
function amountToken0PendingUnstaking() public view override returns
(uint256) {
    uint256 balanceNative = address(this).balance;
    uint256 excessNative =
        balanceNative > amountToken1ClaimableLPWithdrawal ? balanceNative -
amountToken1ClaimableLPWithdrawal : 0;
    // stHYPE is rebase, hence no need for conversion
    uint256 excessToken0 = excessNative > 0 ? excessNative : 0;

    uint256 amountToken0PendingUnstakingCache =
_amountToken0PendingUnstaking;
    if (amountToken0PendingUnstakingCache > excessToken0) {
        return amountToken0PendingUnstakingCache - excessToken0;
    } else {
        return 0;
    }
}
```

However, **amountToken1PendingLPWithdrawal** is not the same, since it doesn't have a getter function and is not updated with the current native balance in the **stHYPEWithdrawalModule** contract. Therefore, when the overseer has already sent native Token1 from unstaking but **update()** hasn't been called, **amountToken1PendingLPWithdrawal** remains outdated, leading to a miscalculation of the total assets in the share calculation.

For example, after some withdrawals are queued, **amountToken1PendingLPWithdrawal** increases to 10e18 from **burnToken0AfterWithdraw**. Then, the owner calls **stHYPEWithdrawalModule::unstakeToken0Reserves** for the first time with **amount = 10e18** to unstake Token0 from the Sovereign pool and burn it via the Overseer contract to claim native Token1 asynchronously.

Before the overseer sends tokens:
amountToken1PendingLPWithdrawal = 10e18,
amountToken0PendingUnstaking = 10e18
=> Total assets = **reserve0 + reserve1 + amountToken1LendingPool + amountToken0PendingUnstaking - amountToken1PendingLPWithdrawal**
= **reserve0 + reserve1 + amountToken1LendingPool + 10e18 - 10e18**

After the overseer sends 4e18 native tokens, but **update()** hasn't been called:
amountToken1PendingLPWithdrawal = 10e18 (still outdated),
amountToken0PendingUnstaking = 4e18 (up-to-date via getter function)
=> Total assets = **reserve0 + reserve1 + amountToken1LendingPool + 4e18 - 10e18**

Therefore, the user's shares will be calculated incorrectly due to the use of the outdated **amountToken1PendingLPWithdrawal**, leading to a loss of shares when depositing.

The same issue exists in the calculation of the withdraw function.

DIRECT THEFT OF SURPLUS BALANCE WHEN UNSTAKING STHYPE

SEVERITY: Critical

PATH:

src/stHYPEWithdrawalModule.sol:update#L431-L475

REMEDIATION:

It is not safe to make direct transfers to the pool if you are unsure whether current execution is inside of a swap callback. It should therefore be verified using the reentrancy switch of the Sovereign Pool, however this is currently not exposed.

One possibility is to add a donate function to the SovereignPool that only works for rebase tokens (donations are possible with transfers already anyway) and has the nonReentrant modifier. That way you can safely donate the tokens back to the pool without the risk.

If modifications to the SovereignPool are not possible, then dummy calls that check whether reverts happen due to reentrancy are needed. This is often used to guard against read-only reentrancy for Balancer or Curve.

STATUS: Fixed

DESCRIPTION:

The stHYPE Withdrawal Module exposes functionality for the owner of the module to unstake stHYPE into HYPE using the Overseer contract. The owner calls **unstakeToken0Reserves**, which in turn transfers stHYPE from the SovereignPool (uses the STEXAMM) and then calls **burnAndRedeemIfPossible** to create a Burn request. The stHYPE is burned immediately and the Burn request can be redeemed later once filled for HYPE.

In the **update** function, any received HYPE from redeemed Burn requests is first allocated to any outstanding LP withdrawal requests after which the surplus is deposited back into the Sovereign Pool directly on lines 471-474:

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

This works because the token is a rebase token and so the Sovereign Pool uses the actual balance as reserve.

However, this mechanism can be exploited to directly steal the entire surplus amount by leveraging the swap callback of the Sovereign Pool.

When calling **swap** on the Sovereign Pool, the user has the option parameter **isSwapCallback**, which can be turned on to pay the **tokenIn** amount inside of the callback in **_handleTokenInTransfersOnSwap**:

```
    function _handleTokenInTransfersOnSwap(
        bool isZeroToOne,
        bool isSwapCallback,
        IERC20 token,
        uint256 amountInUsed,
        uint256 effectiveFee,
        bytes calldata _swapCallbackContext
    ) private {
        uint256 preBalance = token.balanceOf(sovereignVault);

        if (isSwapCallback) {

ISovereignPoolSwapCallback(msg.sender).sovereignPoolSwapCallback(
                address(token),
                amountInUsed,
                _swapCallbackContext
            );
        } else {
            token.safeTransferFrom(msg.sender, sovereignVault,
amountInUsed);
        }

        uint256 amountInReceived = token.balanceOf(sovereignVault) -
preBalance;

        [..]
    }
```

Any balance transferred to the Sovereign Pool during this callback would be counted towards payment for the user. As such, it is possible to call **stHYPEWithdrawalModule.update** inside of the callback, where the surplus amount in HYPE is transferred directly.

As a result, the entire amount can be used by the attacker to swap to stHYPE, without actual paying and thus directly stealing the value from the protocol.

```
    function update() external nonReentrant {
        // Need to ensure that enough native token is reserved for settled
LP withdrawals
        // WARNING: This implementation assumes that there is no slashing
enabled in the LST protocol
        uint256 amountToken1ClaimableLPWithdrawalCache =
amountToken1ClaimableLPWithdrawal;
        if (address(this).balance <= amountToken1ClaimableLPWithdrawalCache)
{
            return;
        }

        // Having a surplus balance of native token means that new unstaking
requests have been fulfilled
        uint256 balanceSurplus = address(this).balance -
amountToken1ClaimableLPWithdrawalCache;
        // stHYPE is rebase, hence no need for conversion
        uint256 balanceSurplusToken0 = balanceSurplus;

        uint256 amountToken0PendingUnstakingCache =
_amountToken0PendingUnstaking;
        if (amountToken0PendingUnstakingCache > balanceSurplusToken0) {
            _amountToken0PendingUnstaking =
amountToken0PendingUnstakingCache - balanceSurplusToken0;
        } else {
            _amountToken0PendingUnstaking = 0;
        }

        // Prioritize LP withdrawal requests
        uint256 amountToken1PendingLPWithdrawalCache =
amountToken1PendingLPWithdrawal;
        if (balanceSurplus > amountToken1PendingLPWithdrawalCache) {
            balanceSurplus -= amountToken1PendingLPWithdrawalCache;
            amountToken1ClaimableLPWithdrawal +=
amountToken1PendingLPWithdrawalCache;
```

```
            cumulativeAmountToken1ClaimableLPWithdrawal +=
amountToken1PendingLPWithdrawalCache;
            amountToken1PendingLPWithdrawal = 0;
        } else {
            amountToken1PendingLPWithdrawal -= balanceSurplus;
            amountToken1ClaimableLPWithdrawal += balanceSurplus;
            cumulativeAmountToken1ClaimableLPWithdrawal += balanceSurplus;
            balanceSurplus = 0;
            return;
        }

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

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

Proof-of-concept:

```
function testPoC() public {
    _addPoolReserves(10 ether, 10 ether);

    uint token0Balance = token0.balanceOf(address(this));
    uint token1Balance = weth.balanceOf(address(this));

    (uint reserve0, uint reserve1) = pool.getReserves();
    console.log("reserve0: %e", reserve0);
    console.log("reserve1: %e", reserve1);

    withdrawalModule.unstakeToken0Reserves(5 ether);
    vm.deal(address(overseer), 5 ether);

    (reserve0, reserve1) = pool.getReserves();
    console.log("reserve0: %e", reserve0);
    console.log("reserve1: %e", reserve1);

    token0.approve(address(stex), 1000 ether);
    weth.approve(address(stex), 1000 ether);
    token0.approve(address(pool), 1000 ether);
    weth.approve(address(pool), 1000 ether);

    SovereignPoolSwapParams memory params;
    params.isZeroToOne = false;
    params.amountIn = 5 ether;
    params.deadline = block.timestamp;
    params.swapTokenOut = address(token0); // stHYPE
    params.recipient = address(this);
    params.isSwapCallback = true;

    (uint256 amountInUsed, uint256 amountOut) = pool.swap(params);
    console.log("amountInUsed: %e", amountInUsed);
    console.log("amountOut: %e", amountOut);

    (reserve0, reserve1) = pool.getReserves();
    console.log("reserve0: %e", reserve0);
    console.log("reserve1: %e", reserve1);
```

```
    console.log("+ stHYPE.balance = %e", token0.balanceOf(address(this)) -
token0Balance);
    console.log("+ wHYPE.balance = %e", weth.balanceOf(address(this)) -
token1Balance);
}

function sovereignPoolSwapCallback(MockStHype token, uint amount, bytes
calldata) external {
    overseer.settleBurn(1);
    withdrawalModule.update();
}
```

VLTS3-3

LOSS OF FUNDS WHEN SETTING A NEW LENDINGMODULE WITH STHYPEWITHDRAWALMODULE::SETPROPOSEDLENDINGMODULE

SEVERITY: High

REMEDIATION:

We recommend to set the withdraw receiver as the SovereignPool, such that the withdrawn tokens would simply be counted as reserve. Afterwards, the owner can re-deposit any amount of assets back into the new lending module using supplyToken1ToLendingPool.

STATUS: Fixed

DESCRIPTION:

When a new **lendingModule** is set via **stHYPEWithdrawalModule::setProposedLendingModule** the wrapped native token is transferred from the current **lendingModule** to the **stHYPEWithdrawalModule** contract:

```
if (address(lendingModule) != address(0)) {
    lendingModule.withdraw(lendingModule.assetBalance(), address(this));
}
```

As a consequence, these funds may be stuck in the **stHYPEWithdrawalModule** contract since they are not transferred to the new lending module (**lendingModuleProposal.lendingModule**) and there is also no other way to transfer the withdrawn tokens.

```
function setProposedLendingModule() external onlyOwner {
    if (lendingModuleProposal.startTimestamp > block.timestamp) {
        revert
stHYPEWithdrawalModule__setProposedLendingModule_ProposalNotActive();
    }

    if (lendingModuleProposal.startTimestamp == 0) {
        revert
stHYPEWithdrawalModule__setProposedLendingModule_InactiveProposal();
    }

    if (address(lendingModule) != address(0)) {
        lendingModule.withdraw(lendingModule.assetBalance(), address(this));
    }

    lendingModule = ILendingModule(lendingModuleProposal.lendingModule);
    delete lendingModuleProposal;
    emit LendingModuleSet(address(lendingModule));
}
```

VLTS3-9

WITHDRAWAL QUEUE PRIORITY BYPASS FOR FEELESS INSTANT WITHDRAWAL

SEVERITY: High

REMEDIATION:

The creation of the LP withdrawal request should only take the order into account, e.g. by keeping a cumulative withdrawn amount that can then be used in claim to ensure that the claimer has the rights to the claimable ETH.

STATUS: Fixed

DESCRIPTION:

When a **Withdrawal** is created, it references the **cumulativeAmountToken1ClaimableLPWithdrawalCheckpoint** pointer to track when a claim is allowed, according to the existing withdrawal queue.

```
  LPWithdrawals[idLPWithdrawal] = LPWithdrawalRequest({
      recipient: _recipient,
      amountToken1: amountToken1.toUint96(),
      cumulativeAmountToken1ClaimableLPWithdrawalCheckpoint:
cumulativeAmountToken1ClaimableLPWithdrawal
  });
```

However, **cumulativeAmountToken1ClaimableLPWithdrawal** only gets updated after a call to the **update** function and only if the HYPE was received from the Overseer. So any withdrawal request made after another one while it's not claimable yet are in the same queue at the same cumulative amount:

```
if (address(this).balance <= amountToken1ClaimableLPWithdrawalCache) {
        amountToken1ClaimableLPWithdrawal = address(this).balance;
        return;
}
```

Moreover, it does not keep withdrawal sizes in mind. If a user withdraws 100 ETH, and the **cumulativeAmountToken1ClaimableLPWithdrawalCheckpoint** is X, they can claim the withdrawal at X + 100.

The following logic allows smaller deposits to be bypass the priority queue and effectively gain feeless instant withdrawals of their stHYPE to HYPE.

```
if (
    cumulativeAmountToken1ClaimableLPWithdrawal
        < request.cumulativeAmountToken1ClaimableLPWithdrawalCheckpoint +
request.amountToken1
) {
    revert stHYPEWithdrawalModule__claim_cannotYetClaim();
}
```

Consider the following situation:

1. User 1 withdraws 10 ETH (**cumulativeAmountToken1ClaimableLPWithdrawalCheckpoint = X**), so they can withdraw at **X + 10**.
2. The burn request is filled and redeemed, the Overseer sends 10 ETH to the Withdrawal Module.
3. User 2 withdraws 1 ETH.
4. Update is now called and fills 10 ETH.
5. User 2 can claim.
6. User 1 can claim but reverts due to not enough funds (only 9 ETH available).

```
    function burnToken0AfterWithdraw(uint256 _amountToken0, address
_recipient)
        external
        override
        onlySTEX
        nonReentrant
    {
        // stHYPE is rebase, hence to need for conversion
        uint256 amountToken1 = _amountToken0;

        amountToken1PendingLPWithdrawal += amountToken1;

        emit LPWithdrawalRequestCreated(idLPWithdrawal, amountToken1,
_recipient);
```

```
        LPWithdrawals[idLPWithdrawal] = LPWithdrawalRequest({
            recipient: _recipient,
            amountToken1: amountToken1.toUint96(),
            cumulativeAmountToken1ClaimableLPWithdrawalCheckpoint:
cumulativeAmountToken1ClaimableLPWithdrawal
        });
        idLPWithdrawal++;
    }
```

Proof-of-concept:

```
function testWithdraw__earlyWithdrawalModule() public {

        address recipient = makeAddr("RECIPIENT");
        address recipient2 = makeAddr("RECIPIENT2");

        _deposit(10e18, recipient);
        token0.mint{value: 10e18}(address(pool));

        // user 2 shares.
        uint256 shares2 = stex.deposit(1e18, 1, block.timestamp,
recipient2);
         // user 1 shares.
        uint256 shares = stex.balanceOf(recipient);


        // prank user 1
        vm.startPrank(recipient);
        // withdraw user 1 shares at X = 0 and is able to withdraw at x+10
now
        (uint256 amount0, uint256 amount1) = stex.withdraw(shares, 0, 0,
block.timestamp, recipient, false, false);
        vm.stopPrank();


        // overseer only returned 5 eth.
        vm.deal(address(withdrawalModule), 5e18);
        withdrawalModule.update();
        vm.warp(block.timestamp + 1);
        vm.roll(block.number + 1);


        // prank user2
        vm.startPrank(recipient2);
        // withdraw after 5 eth was updated.
        (uint256 amount4, uint256 amount3) = stex.withdraw(shares2/2, 0, 0,
block.timestamp, recipient2, false, false);
        vm.stopPrank();
```

```
        // after few withdraws it reaches 10 eth.
        vm.deal(address(withdrawalModule), 10e18);
        withdrawalModule.update();

        // user 2 skips priority que and claims first
        withdrawalModule.claim(1);

        // user 1 cannot claim his funds.
        vm.expectRevert();
        withdrawalModule.claim(0);
    }
```

VLTS3-14

READ-ONLY REENTRANCY IN WITHDRAW CAN LEAD TO SWAP FEE MANIPULATION

SEVERITY: Medium

PATH:

src/STEXAMM.sol:withdraw#L523-L624

REMEDIATION:

It is advisable to use a modifier for view function that are susceptible to read-only reentrancy. The modifier only checks whether the reentrancy guard is active and does not need to set it.

STATUS: Fixed

DESCRIPTION:

In the **withdraw** function for LP shareholders in STEXAMM, the caller can choose to unwrap their wHYPE and receive native HYPE instead. This happens on lines 599-602 and an external call is made to the receiver, giving them execution.

Even though the function itself has the **nonReentrant** modifier, it is still an intermediate state: the **token0** (stHYPE) has already been burned previously using **burnToken0AfterWithdraw** on line 589, while remaining **token1** (wHYPE) is yet to be taken out of the pool using **withdrawLiquidity** on line 611.

Even though the pool itself does not use the **token0/token1** ratio for valuation, it does use this ratio to calculate the swap fee in **STEXRatioSwapFeeModule:getSwapFeeInBips**, which is used in **SovereignPool:swap**.

As such, a user can perform a swap inside of the receive callback, when the ratio of **token0/token1** is significantly lower than normal, and get less swap fees as a result.

The reentrancy guard of **STEXAMM** won't be triggered, because **getLiquidityQuote** is **view** and not marked as **nonReentrant**.

```
    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 (cache.amount1LendingPool > 0) {
          _withdrawalModule.withdrawToken1FromLendingPool(
              cache.amount1LendingPool, _unwrapToNativeToken ? address(this)
: _recipient
          );

          if (_unwrapToNativeToken) {
              IWETH9(token1).withdraw(cache.amount1LendingPool);
              Address.sendValue(payable(_recipient),
cache.amount1LendingPool);
          }
        }

        [..]
    }
```

POTENTIAL DOS CONDITION WHEN TRYING TO SET A NEW PROPOSED LENDING MODULE

SEVERITY: Low

PATH:

src/stHYPEWithdrawalModule.sol#L312

REMEDIATION:

Consider checking the assetBalance() to not being 0 before calling lendingModule.withdraw() inside stHYPEWithdrawalModule::setProposedLendingModule.

For example:

```
if (address(lendingModule) != address(0) && lendingModule.assetBalance() != 0) {
    lendingModule.withdraw(lendingModule.assetBalance(), address(this));
}
```

STATUS: Fixed

DESCRIPTION:

AAVE V3 Pool reverts when withdrawing zero amount:

```
function validateWithdraw(
    DataTypes.ReserveCache memory reserveCache,
    uint256 amount,
    uint256 userBalance
  ) internal view {
    require(amount != 0, Errors.INVALID_AMOUNT);
```

Thus, when updating a lending module (**AaveLendingModule.sol**) with **assetBalance() = 0** via **stHYPEWithdrawalModule::setProposedLendingModule** it will revert:

```
if (address(lendingModule) != address(0)) {
    lendingModule.withdraw(lendingModule.assetBalance(), address(this));
}
```

```
function setProposedLendingModule() external onlyOwner {
    if (lendingModuleProposal.startTimestamp > block.timestamp) {
        revert
stHYPEWithdrawalModule__setProposedLendingModule_ProposalNotActive();
    }

    if (lendingModuleProposal.startTimestamp == 0) {
        revert
stHYPEWithdrawalModule__setProposedLendingModule_InactiveProposal();
    }

    if (address(lendingModule) != address(0)) {
        lendingModule.withdraw(lendingModule.assetBalance(), address(this));
    }

    lendingModule = ILendingModule(lendingModuleProposal.lendingModule);
    delete lendingModuleProposal;
    emit LendingModuleSet(address(lendingModule));
}
```

VLTS3-10

OWNER IS ABLE TO SET NEW FEEPARAMS WITHOUT A TIME LOCK

SEVERITY: Low

PATH:

src/STEXRatioSwapFeeModule.sol:setSwapFeeParams#L155-L192

REMEDIATION:

We would recommend to implement some time lock for the setting of fee parameters as well, to allow for withdrawals to happen, as the fee parameters also affect LPs that do instant withdrawals and not just users that perform swaps.

STATUS: Acknowledged

DESCRIPTION:

The **SwapFeeModule** in **STEXAMM** is set through a proposal of a new **STEXRatioSwapFeeModule** by calling the function **setProposedSwapFeeModule**, however the new fee parameters can be set anytime after the proposal was passed without a time lock.

Typically, a new fee module with the updated fee becomes active after 7 days. However, calling **setSwapFeeParams** allows the owner to change the **STEXRatioSwapFeeModule** at any time, without needing to propose and wait for any time to pass when calling **proposeSwapFeeModule** on the **STEXAMM** contract.

There are some safeguards in the **setSwapFeeParams** function, but it could still allow the owner to set the fee to 50% instantly.

```
    function setSwapFeeParams(
        uint32 _minThresholdRatioBips,
        uint32 _maxThresholdRatioBips,
        uint32 _feeMinBips,
        uint32 _feeMaxBips
    ) external override onlyOwner {
        // Reserve ratio threshold params must be in BIPS
        if (_minThresholdRatioBips >= BIPS) {
            revert
STEXRatioSwapFeeModule__setSwapFeeParams_invalidMinThresholdRatio();
        }
        if (_maxThresholdRatioBips > BIPS) {
            revert
STEXRatioSwapFeeModule__setSwapFeeParams_invalidMaxThresholdRatio();
        }
        if (_minThresholdRatioBips >= _maxThresholdRatioBips) {
            revert
STEXRatioSwapFeeModule__setSwapFeeParams_inconsistentThresholdRatioParams();
        }

        // Fees must be lower than 50% (5_000 bips)
        if (_feeMinBips >= BIPS / 2) {
            revert STEXRatioSwapFeeModule__setSwapFeeParams_invalidFeeMin();
        }
        if (_feeMaxBips >= BIPS / 2) {
            revert STEXRatioSwapFeeModule__setSwapFeeParams_invalidFeeMax();
        }

        if (_feeMinBips > _feeMaxBips) {
            revert
STEXRatioSwapFeeModule__setSwapFeeParams_inconsistentFeeParams();
        }

        feeParams = FeeParams({
            minThresholdRatioBips: _minThresholdRatioBips,
            maxThresholdRatioBips: _maxThresholdRatioBips,
            feeMinBips: _feeMinBips,
            feeMaxBips: _feeMaxBips
        });

        emit SwapFeeParamsSet(_minThresholdRatioBips,
_maxThresholdRatioBips, _feeMinBips, _feeMaxBips);
    }
```

VLTS3-11

AAVE LENDING MODULE COULD BE A SOURCE OF TEMPORARY DOS

SEVERITY: Low

PATH:

src/AaveLendingModule:withdraw#L75-L77

REMEDIATION:

The function supplyToken1ToLendingPool allows the owner to withdraw all liquidity from the SovereignPool and supply it to Aave, we recommend that this function should be capped by a sensible percentage of the total reserves to reduce the chance of Aave blocking withdrawals.

STATUS: Acknowledged

DESCRIPTION:

The STEXAMM leverages supplying token1 (WETH) collateral to a lending platform (Aave) to earn extra yield. The depositing happens manually by the owner, but the withdrawing happens automatically upon a call to **STEXAMM.sol:withdraw** to fulfil the callers withdrawal.

The code assumes that the withdraw function of Aave can always return the provided collateral but this is not the case if the liquidity is being borrowed on Aave. Aave often utilises a utilisation factor of 90%, which means that only 10% of the supplied liquidity can be withdrawn at the time. If a higher amount is requested, it would simply revert.

Since HyperEVM is a new chain, we can assume that the STEXAMM will be a major supplied of wHYPE to Aave on HyperEVM and potentially hold more than 10% of the supplied liquidity. In that case, larger withdrawals could result in funds not being withdrawable and temporary insolvency/DoS.

```
    function withdraw(uint256 _amount, address _recipient) external
onlyOwner {
        pool.withdraw(asset, _amount, _recipient);
    }
```

USING TWO STEP OWNERSHIP TRANSFER APPROACH

SEVERITY: Informational

PATH:

src/AaveLendingModulesol#L4

src/stHYPEWithdrawalModule.sol#L8

src/STEXAMM.sol#L14

src/STEXRatioSwapFeeModule.sol#L4

REMEDIATION:

Consider using an updated and thoroughly audited OpenZeppelin implementation of Ownable2Step in place of Ownable.

STATUS: Acknowledged

DESCRIPTION:

We would recommend using **Ownable2Step** from OpenZeppelin instead of **Ownable** for a safer ownership transfer. **Ownable2Step** helps limit accidental transfers of admin privileges to unintended addresses via its two-step acceptance mechanism.

```
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
```

VLTS3-4

UNUSED ERRORS

SEVERITY: Informational

PATH:

src/STEXAMM.sol#L47

src/stHYPEWithdrawalModule.sol#L58

REMEDIATION:

Remove the unused errors.

STATUS: Fixed

DESCRIPTION:

In the **STEXAMM.sol** and **stHYPEWithdrawalModule.sol** contracts the error **stHYPEWithdrawalModule__cancelLendingModuleProposal_ProposalNotActive()**; and error **STEXAMM__setManagerFeeBips_invalidManagerFeeBips()**; are declared, but they are never used.

```
error STEXAMM__setManagerFeeBips_invalidManagerFeeBips();
error
stHYPEWithdrawalModule__cancelLendingModuleProposal_ProposalNotActive();
```

VLTS3-12

REDUNDANT ZERO CHECK IN AMOUNTTOKENOPENDINGUNSTAKING

SEVERITY: Informational

PATH:

stHYPEWithdrawalModule.sol:amountToken0PendingUnstaking#L219-L232

REMEDIATION:

Simplify line 224 by setting directly:

uint256 excessToken0 = excessNative;

STATUS: Fixed

DESCRIPTION:

On line 224, the variable is set using a conditional zero check for **excessNative**, however the conditional can be simplified to directly setting it to **excessNative**.

For example, if **excessNative** is greater than 0, it is set to **excessNative** and if it is 0, it would still be set to 0.

```
    function amountToken0PendingUnstaking() public view override returns
(uint256) {
        uint256 balanceNative = address(this).balance;
        uint256 excessNative =
            balanceNative > amountToken1ClaimableLPWithdrawal ?
balanceNative - amountToken1ClaimableLPWithdrawal : 0;
        // stHYPE is rebase, hence no need for conversion
        uint256 excessToken0 = excessNative > 0 ? excessNative : 0;

        uint256 amountToken0PendingUnstakingCache =
_amountToken0PendingUnstaking;
        if (amountToken0PendingUnstakingCache > excessToken0) {
            return amountToken0PendingUnstakingCache - excessToken0;
        } else {
            return 0;
        }
    }
```


hexens
×
Valantis