hexens × RUSH

JAN.25

SECURITY REVIEW REPORT FOR RUSH TRADING

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 audit covered the smart contracts of Rush Trading, a protocol where users can deploy new ERC20 tokens and borrow bootstrap liquidity from liquidity providers to instantly create a healthy market.

Our security assessment was a full review of the code differences, spanning a total of 1 week.

During our audit, we have identified 2 high-severity vulnerabilities. One could have allowed a user to bypass a large percentage of the liquidity borrowing fee and the other would have allowed stealing ETH from the sponsor account.

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

Finally, all of our reported issues were either 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/rush-trading/contracts/tree/60504a66a5bbecc3a89e89f74814dc07c5aeed4b

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

https://github.com/rush-trading/contracts/tree/2ba660b19dc5f4828f904d983a38d893d4a300e7

AUDITING DETAILS

STARTED
27.01.2025

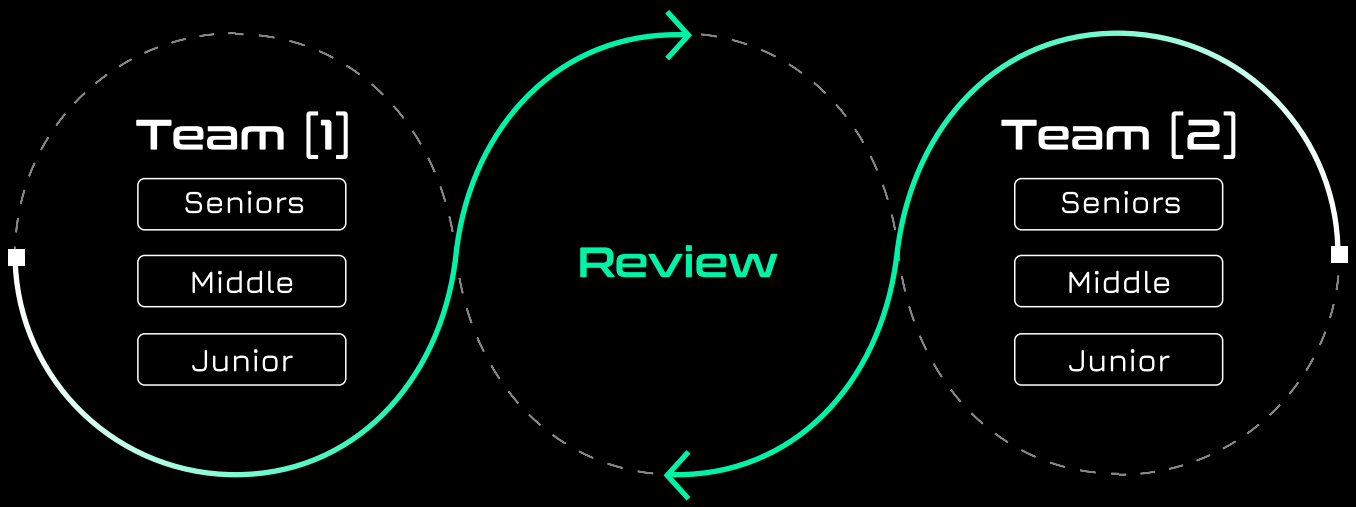
DELIVERED
03.02.2025

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.

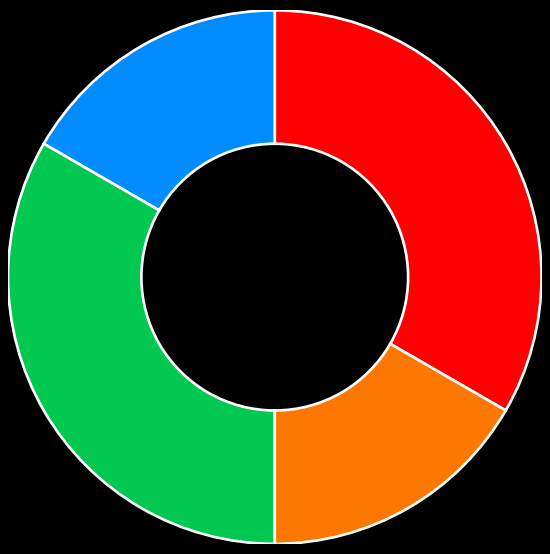
ISSUE SYMBOLIC CODES

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.

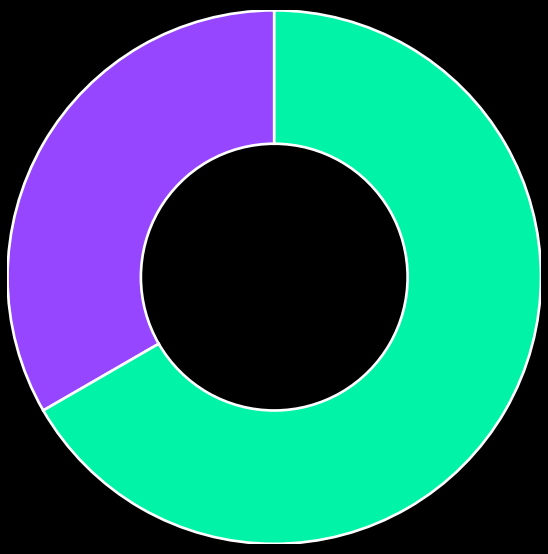
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	2
Medium	1
Low	2
Informational	1

Total: 6



- High
- Medium
- Low
- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

RUSH1-7

RUSHERC20 LAUNCH FEE BYPASS DUE TO FLASH LOAN STAKE

SEVERITY: High

PATH:

LiquidityDeployer.sol:deployLiquidity#L114-L222

REMEDIATION:

Flash loan staking should be protected against by not allowing users to deposit and withdraw in the same block.

STATUS: Acknowledged, see commentary

DESCRIPTION:

When a user launches their own RushERC20, they will have to provide fees as ETH through **msg.value**. The fee consists of a fee paid to the LiquidityPool and a reserve fee that is first deployed as liquidity and later paid to the reserve. In the default parameters, the liquidity pool is 90% and the reserve is 10% of the total fee. The total fee is calculated from the liquidity amount, duration and utilisation ratio of the liquidity pool. The liquidity pool fee can be freely bypassed by any user launching their own RushERC20.

The problem lies in the function **deployLiquidity** where the liquidity pool fee is directly transferred as WETH to the LiquidityPool, where it will be counted as total assets from the WETH balance:

```
// Interactions: Transfer the remaining portion of the fee to the LiquidityPool as
APY.
IERC20(WETH).transfer(LIQUIDITY_POOL, vars.totalFee - vars.reserveFee);
```

The LiquidityPool is a simple ERC4626 staking contract, without locking of assets.

To exploit this vulnerability, an attacker can:

1. Flash loan WETH from any number of DeFi protocols;
2. Deposit this in the LiquidityPool to gain >99.99% of the stake;
3. Launch the RushERC20 which increases the share rate of the LiquidityPool;
4. Withdraw all shares with the higher share rate;
5. Retrieve the paid fees as yield from the LiquidityPool.

This is not only theft of potential yield but it is also a negative impact for the health of the protocol. If there are no fees for providing liquidity, then there is no incentive and worse yet, attackers can forcibly lock liquidity providers' assets for much lower fees.

```
    function deployLiquidity(
        address originator,
        address uniV2Pair,
        address rushERC20,
        uint256 amount,
        uint256 duration,
        uint256 maxTotalFee
    )
        external
        payable
        override
        onlyLauncherRole
        whenNotPaused
    {
        LD.DeployLiquidityLocalVars memory vars;
        // Checks: Pair must not have received liquidity before.
        if (_liquidityDeployments[uniV2Pair].deadline > 0) {
            revert Errors.LiquidityDeployer_PairAlreadyReceivedLiquidity({ rushERC20:
rushERC20, uniV2Pair: uniV2Pair });
        }
        // Checks: Total supply of the RushERC20 token must be greater than 0.
        vars.rushERC20TotalSupply = IERC20(rushERC20).totalSupply();
        if (vars.rushERC20TotalSupply == 0) {
            revert Errors.LiquidityDeployer_TotalSupplyZero({ rushERC20: rushERC20, uniV2Pair:
uniV2Pair });
        }
```

```
        // Checks: Pair should hold entire supply of the RushERC20 token.
        vars.rushERC20BalanceOfPair = IERC20(rushERC20).balanceOf(uniV2Pair);
        if (vars.rushERC20BalanceOfPair != vars.rushERC20TotalSupply) {
            revert Errors.LiquidityDeployer_PairSupplyDiscrepancy({
                rushERC20: rushERC20,
                uniV2Pair: uniV2Pair,
                rushERC20BalanceOfPair: vars.rushERC20BalanceOfPair,
                rushERC20TotalSupply: vars.rushERC20TotalSupply
            });
        }
        // Checks: Amount to deploy must not be less than minimum limit.
        if (amount < MIN_DEPLOYMENT_AMOUNT) {
            revert Errors.LiquidityDeployer_MinLiquidtyAmount(amount);
        }
        // Checks: Amount to deploy must not be greater than maximum limit.
        if (amount > MAX_DEPLOYMENT_AMOUNT) {
            revert Errors.LiquidityDeployer_MaxLiquidtyAmount(amount);
        }
        // Checks: Duration must not be less than minimum limit.
        if (duration < MIN_DURATION) {
            revert Errors.LiquidityDeployer_MinDuration(duration);
        }
        // Checks: Duration must not be greater than maximum limit.
        if (duration > MAX_DURATION) {
            revert Errors.LiquidityDeployer_MaxDuration(duration);
        }
        // Checks: `msg.value` must be at least the liquidity deployment fee.
        (vars.totalFee, vars.reserveFee) = IFeeCalculator(feeCalculator).calculateFee(
            FC.CalculateFeeParams({
                duration: duration,
                newLiquidity: amount,
                outstandingLiquidity: ILiquidityPool(LIQUIDITY_POOL).outstandingAssets(),
                reserveFactor: RESERVE_FACTOR,
                totalLiquidity: ILiquidityPool(LIQUIDITY_POOL).lastSnapshotTotalAssets()
            })
        );
        if (msg.value < vars.totalFee) {
            revert Errors.LiquidityDeployer_FeeMismatch({ expected: vars.totalFee, received:
msg.value });
        }
        // Checks: Maximum total fee must not be exceeded.
        if (vars.totalFee > maxTotalFee) {
            revert Errors.LiquidityDeployer_MaxTotalFeeExceeded({ totalFee: vars.totalFee,
maxTotalFee: maxTotalFee });
        }

        // Effects: Store the liquidity deployment.
        vars.deadline = block.timestamp + duration;
        _liquidityDeployments[uniV2Pair] = LD.LiquidityDeployment({
            amount: amount.toUint208(),
            deadline: vars.deadline.toUint40(),
            isUnwound: false,
            isUnwindThresholdMet: false,
            subsidyAmount: vars.reserveFee.toUint96(),
```

```
            rushERC20: rushERC20,
            originator: originator
        });

        // Interactions: Dispatch asset from LiquidityPool to the pair.
        ILiquidityPool(LIQUIDITY_POOL).dispatchAsset({ to: uniV2Pair, amount: amount });
        // Interactions: Convert received value from ETH to WETH.
        IWETH(WETH).deposit{ value: msg.value }();
        // Interactions: Transfer reserve fee portion to the pair to maintain
`_unwindLiquidity` invariant.
        IERC20(WETH).transfer(uniV2Pair, vars.reserveFee);
        // Interactions: Transfer the remaining portion of the fee to the LiquidityPool as APY.
        IERC20(WETH).transfer(LIQUIDITY_POOL, vars.totalFee - vars.reserveFee);
        // Interactions: Mint LP tokens to the contract.
        IUniswapV2Pair(uniV2Pair).mint(address(this));
        // Interactions: Swap any excess ETH to RushERC20.
        unchecked {
            vars.excessValue = msg.value - vars.totalFee;
        }
        if (vars.excessValue > 0) {
            _swapWETHToRushERC20({ uniV2Pair: uniV2Pair, originator: originator, wethAmountIn:
vars.excessValue });
        }

        // Emit an event.
        emit DeployLiquidity({
            originator: originator,
            rushERC20: rushERC20,
            uniV2Pair: uniV2Pair,
            amount: amount,
            totalFee: vars.totalFee,
            reserveFee: vars.reserveFee,
            deadline: vars.deadline
        });
    }
```

Commentary from the client:

"We acknowledge this is an acceptable risk, as we use RushRouterAlpha in production where all launch functions are gated. A user would have to receive a valid signature from our backend API in order to successfully interact with any of our gated router functions. In the case of such an attack, we can properly adjust the backend to not serve the malicious actor. And in the worst case scenario, pause the token creation backend until all open liquidity positions are unwound."

Proof of concept:

```
    function test_fee_bypass() public {
        uint lpAmount = 1 ether;
        vm.deal(lp, lpAmount);
        vm.startPrank(lp);
        weth.deposit{value: lpAmount}();
        weth.approve(address(liquidityPool), type(uint).max);
        liquidityPool.deposit(lpAmount, address(lp));
        vm.stopPrank();

        // Snapshot
        vm.roll(block.number + 1);
        liquidityPool.deposit(0, address(this));

        // Flashloan
        uint attackerAmount = 1_000_000 ether;
        uint feeAmount = 1759999999923504000;
        vm.deal(address(attacker), attackerAmount + feeAmount);
        vm.startPrank(attacker);

        weth.deposit{value: attackerAmount}();
        weth.approve(address(liquidityPool), type(uint).max);

        // Wrap launch in deposit-withdraw
        liquidityPool.deposit(attackerAmount, attacker);
        router.launchERC20{value: feeAmount}("Test", "Test", 1e18, 1e18, 365 days,
type(uint).max);
        liquidityPool.redeem(liquidityPool.balanceOf(attacker), attacker, attacker);

        vm.stopPrank();

        console.log("weth.balanceOf(attacker)", weth.balanceOf(attacker) -
attackerAmount);
    }
```

UTILISATION RATIO CAN EXCEED 100% TO INFLATE FEES AND EXPLOIT THE SPONSOR

SEVERITY: High

PATH:

FeeCalculator.sol:calculateFee#L55-L84

REMEDIATION:

The utilisation ratio should be calculated using the:

- max(actualTotalLiquidity, snapshotTotalLiquidity) for the sponsored fee calculation in RushRouterAlpha.sol on lines 463-471.
- min(actualTotalLiquidity, snapshotTotalLiquidity) for the fee calculation in LiquidityDeployer.sol on lines 165-173.

STATUS: Fixed

DESCRIPTION:

The **calculateFee** function calculates the required fee for RushERC20 deployment by taking the liquidity amount, duration and utilisation ratio into account.

However, it receives the snapshotted total assets of the LiquidityPool as input for **params.totalLiquidity**. This means that it becomes possible to have a case where **params.outstandingLiquidity + params.newLiquidity > params.totalLiquidity**.

For example, consider the case when:

1. The snapshot of the assets in the liquidity is **1 wei**;
2. Someone deposits **1 ether** into the liquidity pool;
3. In the same block someone launches an ERC20 with **1 ether** as liquidity amount.

The utilisation ratio becomes **1e36** and the resulting fee also becomes much higher than the actual liquidity amount. Launching won't revert, as the liquidity amount is available in the LiquidityPool.

This can be exploited against the sponsored launch of a RushERC20 in the RushRouterAlpha. The sponsor only pays the fees for a deployment of minimal liquidity amount and minimal liquidity duration. The resulting fee is assumed to be low in that case.

But by inflating the utilisation ratio, the sponsor will be paying far too much and lose funds. These funds will be distributed to the liquidity pool, where the attacker could be controlling the pool and gain those assets.

For the default parameters, with a minimum liquidity amount of **0.00001 ether** and minimum liquidity duration of **1 second**, the inflated fee becomes almost **6 ether** for every sponsored launch.

The reverse is also true for the fees of a normal launch. The snapshot of the total liquidity can be made very high for a single block to lower the fees in the next block.

```
    function calculateFee(FC.CalculateFeeParams calldata params)
        external
        view
        override
        returns (uint256 totalFee, uint256 reserveFee)
    {
        FC.CalculateFeeLocalVars memory vars;

        vars.feeRate = BASE_FEE_RATE;
        vars.utilizationRatio =
            Math.mulDiv(params.outstandingLiquidity + params.newLiquidity, 1e18,
params.totalLiquidity);
        if (vars.utilizationRatio > OPTIMAL_UTILIZATION_RATIO) {
            // If U > U_optimal, formula is:
            //                                                          U - U_optimal
            // R_fee = BASE_FEE_RATE + RATE_SLOPE_1 + RATE_SLOPE_2 * ----------------
            //                                                          1 - U_optimal
            vars.feeRate += RATE_SLOPE_1
                + Math.mulDiv(RATE_SLOPE_2, vars.utilizationRatio -
OPTIMAL_UTILIZATION_RATIO, MAX_EXCESS_UTILIZATION_RATIO);
        } else {
            // Else, formula is:
            //                                            U
            // R_fee = BASE_FEE_RATE + RATE_SLOPE_1 *  -----------
            //                                         U_optimal
            vars.feeRate += Math.mulDiv(RATE_SLOPE_1, vars.utilizationRatio,
OPTIMAL_UTILIZATION_RATIO);
        }

        totalFee = Math.mulDiv(vars.feeRate * params.duration, params.newLiquidity,
1e18);
        reserveFee = Math.mulDiv(totalFee, params.reserveFactor, 1e18);
    }
```

LACK OF SLIPPAGE PROTECTION IN WITHDRAW FUNCTIONS

SEVERITY: Medium

PATH:

src/periphery/RushRouter.sol

src/periphery/RushRouterAlpha.sol

REMEDIATION:

Allow users to specify a maximum number of shares to use when withdrawing from LiquidityPool.

STATUS: Fixed

DESCRIPTION:

The **withdraw()** and **withdrawETH()** functions both call **redeem()** on the LiquidityPool. However, these functions don't have a slippage protection feature, meaning the depositor may not use the exact number of shares they expect. The value of shares used is based on the total token balance in the LiquidityPool, which can change, especially if new funds are added. This fluctuation can lead to using more shares than anticipated when withdrawing.

```
    function withdraw(uint256 amount) external {
        // Calculate the amount of shares to redeem.
        uint256 shares = LIQUIDITY_POOL.previewWithdraw(amount);

        // Transfer the amount of shares to this contract.
        IERC20(LIQUIDITY_POOL).transferFrom({ from: msg.sender, to: address(this),
value: shares });

        // Redeem the amount of shares from the LiquidityPool and transfer the
corresponding amount of WETH to the
        // sender.
        LIQUIDITY_POOL.redeem({ shares: shares, receiver: msg.sender, owner:
address(this) });
    }

    /**
     * @notice Withdraw lent ETH from the LiquidityPool.
     * @param amount The amount of ETH to withdraw.
     */
    function withdrawETH(uint256 amount) external {
        // Calculate the amount of shares to redeem.
        uint256 shares = LIQUIDITY_POOL.previewWithdraw(amount);

        // Transfer the amount of shares to this contract.
        IERC20(LIQUIDITY_POOL).transferFrom({ from: msg.sender, to: address(this),
value: shares });

        // Redeem the amount of shares from the LiquidityPool and transfer the
corresponding amount of WETH to this
        // contract.
        uint256 received = LIQUIDITY_POOL.redeem({ shares: shares, receiver:
address(this), owner: address(this) });

        // Withdraw the received WETH to ETH.
        IWETH(WETH).withdraw(received);

        // Transfer the amount of WETH to the sender.
        payable(msg.sender).transfer(received);
    }
```

REWARDS IN STAKINGREWARDS ARE LOST IF TOTAL SUPPLY IS ZERO

SEVERITY: Low

PATH:

StakingRewards.sol:rewardPerToken#L90-L96

REMEDIATION:

We see two options:

1. Introduce a sweep function that can only be called by an authorised role and only after the staking period has finished.
2. Divide the unaccounted token amounts linearly over the remaining duration and add to the rewardRate.

STATUS: Acknowledged, see commentary

DESCRIPTION:

The function **rewardPerToken** calculates the current amount of reward tokens per share inside of the StakingRewards. It is calculated using the initially set **rewardRate** and the delta time of **lastTimeRewardApplicable()** and **lastUpdateTime**. However, it also has an edge case if the total supply is zero, where it directly returns the **rewardPerTokenStored**.

This function is called in the modifier **updateReward** where the **lastUpdateTime** is also set to the current applicable time. As a result, the rewards that were accumulated inside of that time frame would be lost and unaccounted for.

It is correct that the next depositing user would not get these rewards, but now those tokens would be stuck in the contract and lost forever.

This case will almost certainly happen at the start of the StakingRewards, where there would be some time without any supply, while the reward rate immediately starts.

```
function rewardPerToken() public view override returns (uint256) {
    if (totalSupply == 0) {
        return rewardPerTokenStored;
    }
    return rewardPerTokenStored
        + Math.mulDiv((lastTimeRewardApplicable() - lastUpdateTime) * rewardRate,
1e18, totalSupply);
}
```

Commentary from the client:

"We've decided to accept this limitation after careful evaluation. We find potential remediation approaches would introduce disproportionate complexity, making this an acceptable tradeoff that aligns with our current objectives."

_GETISUNWINDTHRESHOLDMET SHOULD INCLUDE SUBSIDYAMOUNT FOR THRESHOLD CALCULATION

SEVERITY: Low

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

To unwind liquidity before the deadline, the current reserves must exceed the **EARLY_UNWIND_THRESHOLD**. This threshold should also include the **subsidyAmount**, since the **initialWETHReserve** is the sum of **deployment.amount** and **deployment.subsidyAmount**.

```
    function _getIsUnwindThresholdMet(address uniV2Pair) internal view
returns (bool isUnwindThresholdMet) {
        LD.LiquidityDeployment storage deployment =
_liquidityDeployments[uniV2Pair];
        (uint256 currentReserve,,) = _getOrderedReserves(uniV2Pair);
        uint256 targetReserve = deployment.amount + EARLY_UNWIND_THRESHOLD;
        isUnwindThresholdMet = currentReserve >= targetReserve;
    }
```

```
    function _getIsUnwindThresholdMet(address uniV2Pair) internal view
returns (bool isUnwindThresholdMet) {
        LD.LiquidityDeployment storage deployment =
_liquidityDeployments[uniV2Pair];
        (uint256 currentReserve,,) = _getOrderedReserves(uniV2Pair);
--      uint256 targetReserve = deployment.amount + EARLY_UNWIND_THRESHOLD;
++      uint256 targetReserve = deployment.amount + deployment.subsidyAmount
EARLY_UNWIND_THRESHOLD;
        isUnwindThresholdMet = currentReserve >= targetReserve;
    }
```

GASMODE ENUM FUNCTIONALITY NOT ALIGNED WITH ITS NAME

SEVERITY: Informational

REMEDIATION:

Change the enum name into a name more aligned with its purpose, such as launchtype or use the ERC20Type enum and add .Sponsored

STATUS: Fixed

DESCRIPTION:

The enum **GasMode** is suggesting gas would be covered by the sponsor when calling **launchERC20Sponsored** because of the 2 difference options, **Default** or **Sponsored**. Yet the code flow suggest otherwise, currently its just validating the signature for the default or sponsored launch type.

```
// Check the ECDSA signature is valid.
  _checkSignature({
    message: abi.encodePacked(
    msg.sender,
    _useNonce(msg.sender),
    address(this),
    liquidityAmount,
    liquidityDuration,
    ERC20Type.Basic,
    gasMode
    ),
.   signature: signature
  });
```


hexens
×
RUSH