hexens × RUSH

SEPT.24

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 reviewed the contracts from the Rush Trading Protocol, which allows the deployer to create RushERC20 tokens and set up a Uniswap V2 pair between RushERC20 and WETH tokens sourced from the liquidity pool. The deployer is charged a WETH fee, which is then redeposited into the liquidity pool as a reward for depositors.

Our security assessment involved a thorough review of all smart contracts, conducted over the course of one week.

During our audit, we identified one high-severity vulnerability that allows the deployer to lower their fee when launching a new Uniswap V2 pair using a flash loan. Additionally, we found five medium-severity issues, two low-severity issue, and two informational issues.

Finally, all of our reported issues were fixed and 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-clone/tree/a3fbfa370c34fc09978d7762dead3412bfaadfc6

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

https://github.com/rush-trading/contracts-clone/tree/8fcdd46cf870d8dbc1e3061429bb6de5b2819700

AUDITING DETAILS

STARTED
16.09.2024

DELIVERED
23.09.2024

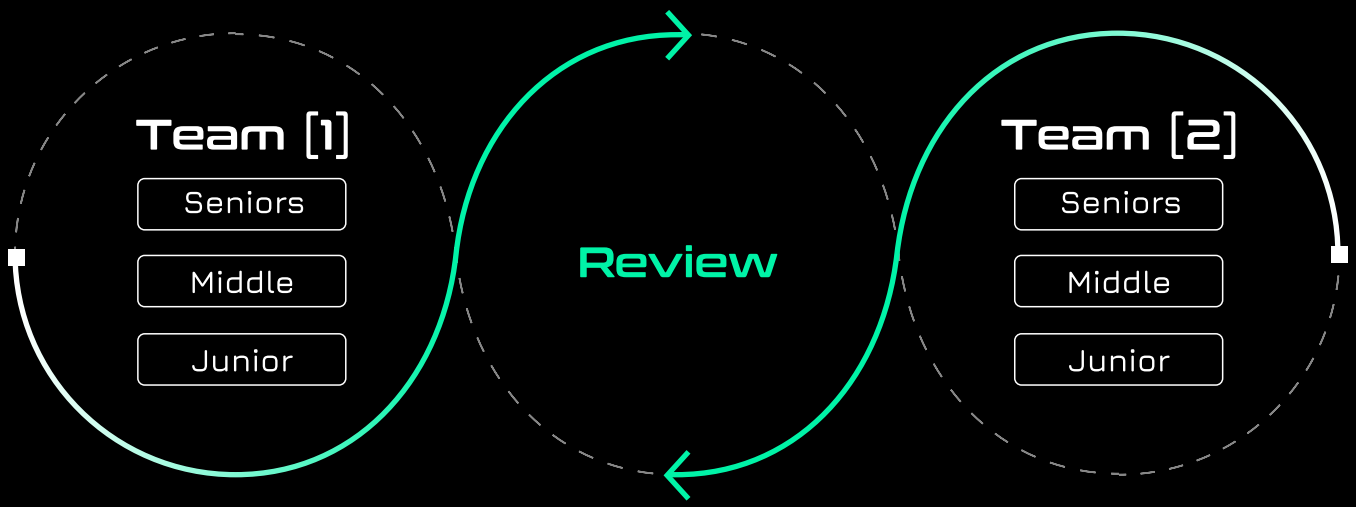
Review Led by

TRUNG DINH

Senior Security Researcher | 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.

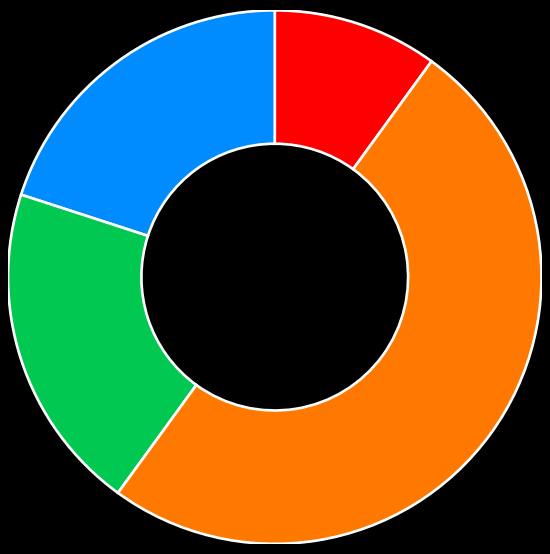
ISSUE SYMBOLIC CODES

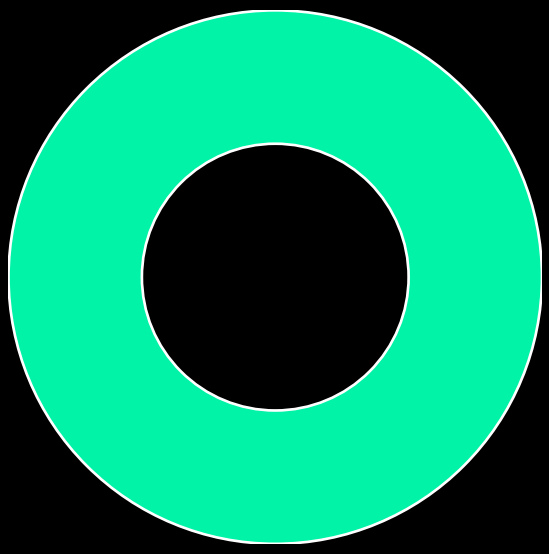
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	5
Low	2
Informational	2

Total: 10





WEAKNESSES

This section contains the list of discovered weaknesses.

RSHTR-5

USERS CAN LOWER FEES USING FLASH LOANS WHEN LAUNCHING A NEW POOL

SEVERITY: High

PATH:

src/LiquidityDeployer.sol#L174-L182

src/FeeCalculator.sol#L69-L82

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

When launching a new pool for WETH and rushERC20 using the **LiquidityDeployer.deployLiquidity()** function, the deployer must pay a fee of **vars.totalFee** in ETH. This fee is calculated by the **FeeCalculator.calculateFee()** function, which can be found between lines 174-182 of the **LiquidityDeployer** contract.

The **FeeCalculator.calculateFee()** function uses the following inputs:

- **newLiquidity**: the amount of WETH transferred from the liquidity pool to deploy the new pair
- **outstandingLiquidity**: the amount of WETH already used to launch pairs

- **totalLiquidity**: the current **totalAssets()** of the liquidity pool

The function calculates the utilization rate **vars.utilizationRatio** using this formula:

```
vars.utilizationRatio = (outstandingLiquidity + newLiquidity) /
totalLiquidity
```

According to the fee rate determination logic (lines 69-82 of the **FeeCalculator** contract), a lower **vars.utilizationRatio** results in a lower fee rate.

```
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);
}
```

This introduces a vulnerability that an attacker could exploit to reduce fees by minimizing the **utilizationRatio**. This can be achieved by temporarily inflating the **totalLiquidity** value before launching a new pool.

For example, an attacker could take out a large WETH flash loan from a pool (e.g., Uniswap V2's WETH-USDC pool) and deposit the borrowed amount into the liquidity pool. This would increase the **totalAssets()** value of the liquidity pool, thus inflating **totalLiquidity**. When the attacker launches a new pool, the high **totalLiquidity** results in a low fee rate (due to a low utilization ratio). After deploying the pool with reduced fees, the attacker can redeem their shares from the liquidity pool and repay the flash loan.

```
    (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).totalAssets()
        })
    );
```

```
    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);
    }
```

Consider introducing a new function, **takeTotalAssetsSnapshot()**, within the Liquidity Pool contract, which allows taking a snapshot of the **totalAssets**. This function should only be callable by a restricted role, such as an admin.

Example:

```
function takeTotalAssetsSnapshot() external onlyAdminRole {
  lastSnapshotTotalAssets = totalAssets();
  lastSnapshotTimestamp = block.timestamp;
}
```

The **lastSnapshotTotalAssets** will be used as the **totalLiquidity** in the **LiquidityDeployer.deployLiquidity()** function, replacing the use of the current **totalAssets()** value. This approach prevents temporary inflation of the liquidity pool's total assets.

For this solution to be effective, the **takeTotalAssetsSnapshot()** function needs to be invoked regularly to keep the **totalAssets()** value updated. Additionally, it would be beneficial to enforce a restriction in the **LiquidityDeployer.deployLiquidity()** function, ensuring that the snapshot is recent enough. This could be done by adding a requirement for the timestamp of the snapshot, as shown below:

```
function deployLiquidity(
    address originator,
    address uniV2Pair,
    address rushERC20,
    uint256 amount,
    uint256 duration
)
    external
    payable
    override
    onlyLauncherRole
    whenNotPaused
{
    uint256 lastSnapshotTimestamp =
ILiquidityPool(LIQUIDITY_POOL).lastSnapshotTimestamp();
```

```
    if (block.timestamp - lastSnapshotTimestamp < MAX_SNAPSHOT_DURATION) {
        revert Errors.Outdated_Snapshot;
    }

    ...
}
```

This ensures that liquidity is only deployed using a recent snapshot of the pool's assets.

RSHTR-1

USERS CAN BYPASS THE MAXIMUM DEPOSIT LIMIT OF THE LIQUIDITYPOOL CONTRACT

SEVERITY: Medium

PATH:

src/LiquidityPool.sol#L130-L133

REMEDIATION:

Consider adding the max total deposit check for the mint() function.

STATUS: Fixed

DESCRIPTION:

The **LiquidityPool.deposit()** function includes a check to ensure that the total assets do not exceed the predefined **maxTotalDeposits** after a deposit. However, since the **LiquidityPool** contract inherits from the **ERC4626** contract, which includes an alternative deposit method, the **mint()** function, users can exploit this by depositing assets via **mint()** without being constrained by the total asset limit enforced by the **deposit()** function.

```
// Checks: total deposits must not exceed the maximum limit.
if (assets + totalAssets() > maxTotalDeposits) {
    revert Errors.LiquidityPool_MaxTotalDepositsExceeded();
}
```

RSHTR-3

THE `LAUNCH()` FUNCTION OF RUSHLAUNCHER CAN BE DOS ATTACKED

SEVERITY: Medium

PATH:

src/RushLauncher.sol#L92-L96

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the **RushLauncher::launch()** function, it deploys a new **rushERC20** token using the **createRushERC20()** function of the **RushERC20Factory** contract. The address of the deployed contract is predictable and can be pre-calculated by front-running and simulating the transaction.

```
rushERC20 =
            IRushERC20Factory(RUSH_ERC20_FACTORY).createRushERC20({ originator:
params.originator, kind: params.kind });
```

```
//src/RushERC20Factory.sol#L62-L64
function createRushERC20(bytes32 kind, address originator) external onlyLauncherRole
returns (address rushERC20) {
        // Effects: Create a new token using the implementation.
        rushERC20 = _templates[kind].clone();
```

After that, the **launch()** function attempts to create a UniswapV2 pair for the deployed RushERC20 token. However, this action can revert if a pair for these tokens already exists.

```
uniV2Pair = IUniswapV2Factory(UNISWAP_V2_FACTORY).createPair({ tokenA:
rushERC20, tokenB: WETH });
```

```
//https://github.com/Uniswap/v2-core/blob/master/contracts/
UniswapV2Factory.sol#L23-L27
function createPair(address tokenA, address tokenB) external returns
(address pair) {
        require(tokenA != tokenB, 'UniswapV2: IDENTICAL_ADDRESSES');
        (address token0, address token1) = tokenA < tokenB ? (tokenA,
tokenB) : (tokenB, tokenA);
        require(token0 != address(0), 'UniswapV2: ZERO_ADDRESS');
        require(getPair[token0][token1] == address(0), 'UniswapV2:
PAIR_EXISTS'); // single check is sufficient
```

Therefore, an attacker can carry out a DoS attack on the **launch()** function by front-running, pre-calculating the address of the **RushERC20** token, and creating a UniswapV2 pair with that address.

```
function launch(
        RL.LaunchParams calldata params
    )
        external
        payable
        override
        onlyRouterRole
        returns (address rushERC20, address uniV2Pair)
    {
        // Checks: Maximum supply must be greater than the minimum limit.
        if (params.maxSupply < MIN_SUPPLY_LIMIT) {
            revert Errors.RushLauncher_LowMaxSupply(params.maxSupply);
        }
        // Checks: Maximum supply must be less than the maximum limit.
        if (params.maxSupply > MAX_SUPPLY_LIMIT) {
            revert Errors.RushLauncher_HighMaxSupply(params.maxSupply);
        }

        // Interactions: Create a new RushERC20 token.
        rushERC20 =
            IRushERC20Factory(RUSH_ERC20_FACTORY).createRushERC20({
originator: params.originator, kind: params.kind });
        // Interactions: Create the Uniswap V2 pair.
        uniV2Pair = IUniswapV2Factory(UNISWAP_V2_FACTORY).createPair({
tokenA: rushERC20, tokenB: WETH });
        // Interactions: Initialize the RushERC20 token.
        IRushERC20(rushERC20).initialize({
            name: params.name,
            symbol: params.symbol,
            maxSupply: params.maxSupply,
            recipient: uniV2Pair,
            data: params.data
        });
        // Interactions: Deploy the liquidity.
        ILiquidityDeployer(LIQUIDITY_DEPLOYER).deployLiquidity{ value:
msg.value }({
            originator: params.originator,
            uniV2Pair: uniV2Pair,
            rushERC20: rushERC20,
            amount: params.liquidityAmount,
            duration: params.liquidityDuration
        });
```

```
        // Emit an event.
        emit Launch({
            rushERC20: rushERC20,
            kind: params.kind,
            uniV2Pair: uniV2Pair,
            maxSupply: params.maxSupply,
            liquidityAmount: params.liquidityAmount,
            liquidityDuration: params.liquidityDuration
        });
    }
```

Add a check to skip the createPair() call if the pair already exists.

```
// Interactions: Create the Uniswap V2 pair.
uniV2Pair = IUniswapV2Factory(UNISWAP_V2_FACTORY).getPair({ tokenA:
rushERC20, tokenB: WETH });
if (uniV2Pair == address(0)) {
    uniV2Pair = IUniswapV2Factory(UNISWAP_V2_FACTORY).createPair({ tokenA:
rushERC20, tokenB: WETH });
}
```

RSHTR-4

DOS IN THE RUSHROUTER.WITHDRAWETH() FUNCTION DUE TO MISSING RECEIVE() FUNCTION

SEVERITY: Medium

PATH:

src/periphery/RushRouter.sol#L203-L204

src/periphery/RushRouterAlpha.sol#L235-L236

REMEDIATION:

Adding the receive() function to the Push router contract.

STATUS: Fixed

DESCRIPTION:

The **RushRouter.withdrawETH()** function is designed to redeem shares from the liquidity pool, convert WETH to ETH, and send the ETH to the user. During the process, it calls **IWETH(WETH).withdraw(received)** to convert the received WETH to ETH. However, the contract lacks a **receive()** function, which prevents it from accepting the ETH sent by the WETH contract. As a result, the transaction reverts, and the user is unable to receive their ETH.

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

RSHTR-10

LACK OF SLIPPAGE PROTECTION IN DEPOSIT FUNCTION

SEVERITY: Medium

PATH:

src/periphery/RushRouter.sol#L147-L170

src/periphery/RushRouterAlpha.sol#L179-L202

REMEDIATION:

Allow users to specify a minimum number of received shares in RushRouter contract when depositing into LiquidityPool.

STATUS: Fixed

DESCRIPTION:

In RushRouter contract, **lend()** and **lendETH()** function attempt to call **deposit()** function of the LiquidityPool contract. However, they lack a slippage protection mechanism to ensure the depositor receives the expected amount of shares. The **totalAsset()** function of LiquidityPool is derived from the token balance of the contract, which can cause fluctuations in the value of the shares. As a result, the **deposit()** function may result in fewer shares than expected when the LiquidityPool receives additional funds, such as rewards from the reserve.

```
function lend(uint256 amount) external {
    // Transfer the amount of WETH to this contract.
    IERC20(WETH).transferFrom({ from: msg.sender, to: address(this), value: amount });

    // Approve the LiquidityPool to spend the WETH.
    IERC20(WETH).approve({ spender: address(LIQUIDITY_POOL), value: amount });

    // Deposit the WETH into the LiquidityPool and mint the corresponding amount of shares to the sender.
    LIQUIDITY_POOL.deposit({ assets: amount, receiver: msg.sender });
}

/**
  * @notice Lend ETH to the LiquidityPool.
  */
function lendETH() external payable {
    // Deposit the ETH into WETH.
    IWETH(WETH).deposit{ value: msg.value }();

    // Approve the LiquidityPool to spend the WETH.
    IERC20(WETH).approve({ spender: address(LIQUIDITY_POOL), value: msg.value });

    // Deposit the WETH into the LiquidityPool and mint the corresponding amount of shares to the sender.
    LIQUIDITY_POOL.deposit({ assets: msg.value, receiver: msg.sender });
}
```

RSHTR-11

UNABLE TO UNWIND LIQUIDITY FROM THE UNISWAP V2 PAIR DUE TO INSUFFICIENT LIQUIDITY MINTED WHEN RESUPPLYING TOKENS

SEVERITY: Medium

PATH:

src/LiquidityDeployer.sol#L406-L414

REMEDIATION:

Consider avoiding the minting of shares if the calculated rushERC20ToResupply is zero. Additionally, we recommend pre-calculating the minted shares (by simulating the Uniswap v2 pool calculation in the mint() function) to prevent the zero minted shares issue, which could otherwise result in a reverted transaction.

STATUS: Fixed

DESCRIPTION:

In the **LiquidityDeployer._unwindLiquidity()** function, if **vars.wethBalance** exceeds **vars.initialWETHReserve**, a portion of the surplus WETH and a corresponding amount of RUSH ERC20 tokens are resupplied to the Uniswap v2 pair. The amount of RUSH ERC20 tokens is calculated using the round-down operation: **vars.rushERC20ToResupply = Math.mulDiv(vars.rushERC20Balance, vars.wethToResupply, vars.wethBalance);**, which may result in a zero value. As a result, the calculated liquidity minted in the **mint()** function of the Uniswap v2 pair could be zero, leading to the error "UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED" and violating the invariant described in the comment from lines 375 to 376.

```
        vars.wethToResupply = vars.wethSurplus - vars.wethSurplusTax;
        // Calculate the amount of RushERC20 to resupply to the pair.
        vars.rushERC20ToResupply = Math.mulDiv(vars.rushERC20Balance,
vars.wethToResupply, vars.wethBalance);
        // Interactions: Transfer the WETH to resupply to the pair.
        IERC20(WETH).transfer(uniV2Pair, vars.wethToResupply);
        // Interactions: Transfer the RushERC20 to resupply to the pair.
        IERC20(deployment.rushERC20).transfer(uniV2Pair,
vars.rushERC20ToResupply);
        // Interactions: Mint LP tokens and send them to the RushERC20 token
address to lock them forever.
        IUniswapV2Pair(uniV2Pair).mint(deployment.rushERC20);
```

Here is a test where the **unwindLiquidity** call is reverted by the **UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED** error. You can add it to the **unwindLiquidity.t.sol** test file, adjust **RUSH_ERC20_SUPPLY** to 1e18 in **test/utils/Defaults.sol**, and then run it with the command **forge test -vv --mt test_unwindLiquidityZeroMintingRevert**:

```
function test_unwindLiquidityZeroMintingRevert()
    external
    whenContractIsNotPaused
    givenPairHasReceivedLiquidity
    givenPairHasNotBeenUnwound
    givenDeadlineHasPassedButEarlyUnwindThresholdIsNotReached
{
    // Set time to be at the deadline.
    LD.LiquidityDeployment memory liquidityDeployment =
liquidityDeployer.getLiquidityDeployment(uniV2Pair);
    vm.warp(liquidityDeployment.deadline);

    // Send WETH amount to the pair to trigger the surplus condition.
    uint256 wethAmount = 1660;
    (, address caller,) = vm.readCallers();
    resetPrank({ msgSender: users.sender });
    deal({ token: address(weth), to: users.sender, give: wethAmount });
    weth.transfer(uniV2Pair, wethAmount);
    resetPrank({ msgSender: caller });

    // Unwind the liquidity and gracefully handle `IUniswapV2Pair.mint`
revert with `IUniswapV2Pair.sync`.
    uint256 liquidityPoolWETHBalanceBefore =
weth.balanceOf(address(liquidityPool));
    liquidityDeployer.unwindLiquidity({ uniV2Pair: uniV2Pair });
    uint256 liquidityPoolWETHBalanceAfter =
weth.balanceOf(address(liquidityPool));
}
```

RSHTR-12

VERIFIER_ADDRESS ASSIGNMENT MISSING IN THE CONSTRUCTOR OF RUSHROUTERALPHA CONTRACT

SEVERITY: Low

PATH:

src/periphery/RushRouterAlpha.sol#L76-L81

REMEDIATION:

Consider assigning the value of VERIFIER_ADDRESS in the constructor of RushRouterAlpha contract

STATUS: Fixed

DESCRIPTION:

In the **RushRouterAlpha** contract, the **VERIFIER_ADDRESS** is an immutable variable intended to store the ECDSA verifier address, which has the authority to sign deployment messages. However, the constructor does not assign a value to this variable, leaving **VERIFIER_ADDRESS** as **address(0)**. As a result, any transaction attempting to deploy a RushERC20 contract will revert.

```
constructor(IRushLauncher rushLauncher_) {
    RUSH_LAUNCHER = rushLauncher_;
    LIQUIDITY_DEPLOYER =
ILiquidityDeployer(rushLauncher_.LIQUIDITY_DEPLOYER());
    LIQUIDITY_POOL = ILiquidityPool(LIQUIDITY_DEPLOYER.LIQUIDITY_POOL());
    WETH = LIQUIDITY_DEPLOYER.WETH();
}
```

Note: The issue was found by the Rush internal team, and validated by Hexens.

MISSING REMOVAL OF EXEMPTION FOR THE PREVIOUS OWNER OF THE RUSHERC20TAXABLE TOKEN WHEN CHANGING OWNERSHIP

SEVERITY: Low

PATH:

src/abstracts/ERC20TaxableUpgradeable.sol#L209-L213

REMEDIATION:

Consider removing the old owner out of the tax-exempt addresses.

STATUS: Fixed

DESCRIPTION:

In the **_transferOwnership** function of the ERC20TaxableUpgradeable contract, exemption permission is granted to the new owner to avoid tax charges. However, it doesn't remove the exemption for the previous owner. As a result, after the ownership of the RushERC20Taxable token is transferred, the previous owner can still avoid tax fees on transfers.

```
function _transferOwnership(address newOwner) internal virtual override {
    taxBeneficiary = newOwner;
    _addExemption(newOwner);
    super._transferOwnership(newOwner);
}
```

RSHTR-6

USE UNCHECKED WHEN IT IS SAFE

SEVERITY: Informational

PATH:

src/LiquidityDeployer.sol#L209
src/LiquidityDeployer.sol#L396-L406

REMEDIATION:

Using unchecked in the arithmetic operation improves gas efficiency since the greater-than condition ensures that no underflow or overflow can occur.

STATUS: Fixed

DESCRIPTION:

In Solidity (**^0.8.0**), adding the **unchecked** keyword around arithmetic operations can reduce gas usage where underflow or overflow is impossible due to preceding checks.
In the **LiquidityDeployer::deployLiquidity()** function, there is a check to ensure **msg.value >= vars.totalFee** from line 183 to 185, so the subtraction **vars.excessValue = msg.value - vars.totalFee;** can be placed inside **unchecked**. Similarly, in the **LiquidityDeployer::_unwindLiquidity()** function, the calculations can be placed inside **unchecked** when **vars.wethBalance > vars.initialWETHReserve**.

```
vars.excessValue = msg.value - vars.totalFee;
```

```
vars.initialWETHReserve = deployment.amount + deployment.subsidyAmount;
// If the WETH balance is greater than the initial reserve, the pair has a
surplus.
if (vars.wethBalance > vars.initialWETHReserve) {
    // Calculate the surplus.
    vars.wethSurplus = vars.wethBalance - vars.initialWETHReserve;
    // Tax the surplus to the reserve.
    vars.wethSurplusTax = Math.mulDiv(vars.wethSurplus, RESERVE_FACTOR,
1e18);
    // Calculate the total reserve fee.
    vars.totalReserveFee = deployment.subsidyAmount + vars.wethSurplusTax;
    // Calculate the amount of WETH to resupply to the pair.
    vars.wethToResupply = vars.wethSurplus - vars.wethSurplusTax;
```

OPTIMIZE `LIQUIDITYDEPLOYER.DEPLOYLIQUIDITY()` BY DEPOSITING ALL ETH TO WETH IN A SINGLE TRANSACTION TO SAVE GAS

SEVERITY: Informational

PATH:

src/LiquidityDeployer.sol#L200-L201

src/LiquidityDeployer.sol#L354-L357

REMEDIATION:

Consider depositing all the msg.value to the WETH within 1 transaction.

STATUS: Fixed

DESCRIPTION:

In the **LiquidityDeployer.deployLiquidity()** function, the **msg.value** is currently split into two parts: **vars.totalFee** and **vars.excessValue**. The **vars.totalFee** is deposited into the Uniswap V2 pair to receive liquidity tokens, while the **vars.excessValue** is swapped from ETH to the RUSH ERC20 token in the pair. Since both values require depositing into the **WETH** contract before transferring to the pair, it would be more gas-efficient to deposit the entire **msg.value** into the **WETH** contract at once, instead of making two separate deposits at line 203 and line 355.

```
    // Interactions: Convert received fee from ETH to WETH.
    IWETH(WETH).deposit{ value: vars.totalFee }();
```

```
    // Interactions: Convert ETH to WETH.
    IWETH(WETH).deposit{ value: ethAmountIn }();
    // Interactions: Transfer WETH to the pair.
    IERC20(WETH).transfer(uniV2Pair, ethAmountIn);
```


hexens
×
RUSH