Aug.23

SECURITY REVIEW REPORT FOR ROCIFI

CONTENTS

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.

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date	Audit Completion Date
31.07.2023	07.08.2023

hexens × RociFi

METHODOLOGY

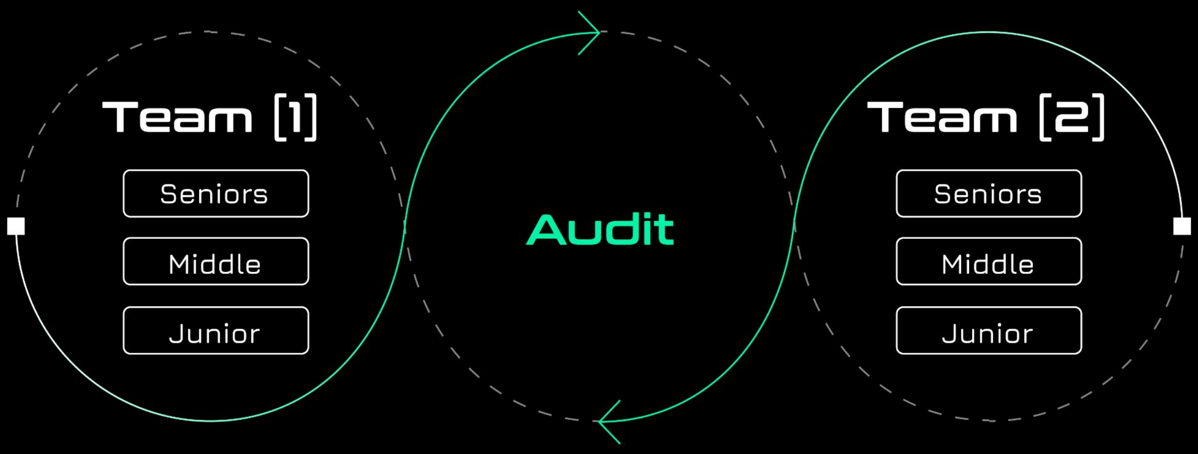
COMMON AUDIT PROCESS

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 a new version of RociFi protocol that will be deployed to the Fuse blockchain. This version very similar to the previous version, expect that the scoring mechanism was removed and that a DIA and Redstone oracle had been added instead of Chainlink.

Our security assessment was a full review of this new version, spanning a total of 1 week.

During our audit, we have identified 3 Critical severity vulnerabilities. One of these vulnerabilities would allow an any user to employ flash-staking and bypass paying interest on their loans.

We have also identified 4 High severity vulnerabilities, 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 of the project have increased after completion of our audit.

SCOPE

The analyzed resources were sent in an archive with the following SHA256 hash:
677aec249d85d7da9d9180f4a07d49700f6a341efaa8672a57c87c7ab6469b00

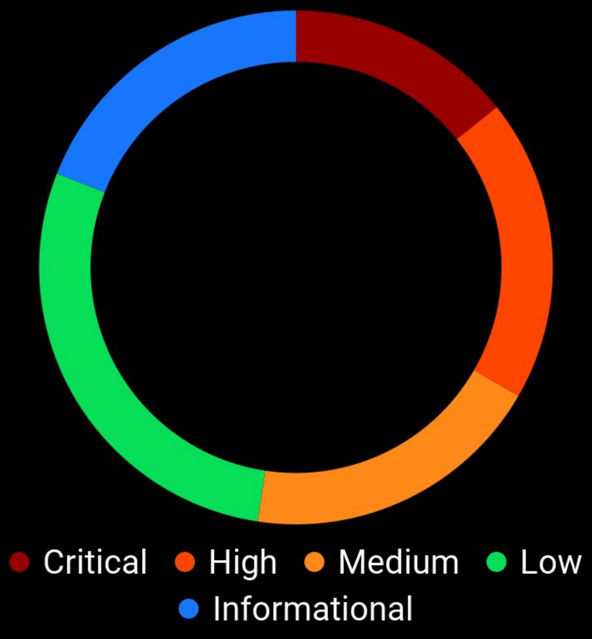
The issues described in the report were fixed in the following version (SHA256 hash):
3cd84018a4f8cdf72acb07d4412738517 85fe7a904e2858aab80faaeb8427945

SUMMARY

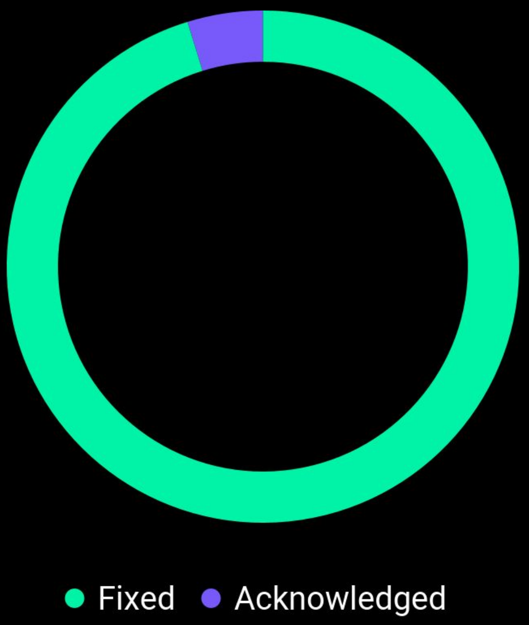
SEVERITY	NUMBER OF FINDINGS
CRITICAL	3
HIGH	4
MEDIUM	4
LOW	6
INFORMATIONAL	4

TOTAL: 21

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

RFIN-4. POOL LOCK-UP PERIOD BYPASS TO STEAL INTEREST OF LOANS

SEVERITY: Critical

PATH: Pool.sol:deposit, repay (L119-136, L143-161)

REMEDIATION: in a staking pool rewards should be emitted in a linear fashion and updated whenever the total supply (e.g. through deposit/withdraw) changes so that no one can steal rewards by depositing large amounts

similarly, the lock up period will not be enough to protect against this. Users with large amount of funds can still temporarily deposit funds to pay less interest

the pool should keep track of the total funds that are borrowed and the combined APR to correctly calculate the outstanding (and thus the pool's value) at any point in time. This would then show a correct value of a Pool's share rate

STATUS: fixed

DESCRIPTION:

When a loan is repaid, the interest payments are transferred from the user to the LoanManager and then to the Pool. Afterwards, the LoanManager also increases the Pool's value to increase the share rate using **Pool:increasePoolValue**.

An attacker can utilise flash-staking to obtain a large amount of shares as possible for a brief transaction. They can then repay their loan and withdraw the large amount of shares to obtain almost all of the paid interest.

The Pool has a last deposit timestamp to prevent people from flash-staking, but this can be bypassed by transferring the large amount of minted shares to a different account and withdrawing them there.

For example:

1. Flash-loan the pool's underlying token
2. Deposit the assets into the pool to obtain the largest share (e.g. 99% shares)
3. Repay the loan (own or sandwich without flash-loan) the increase the pool value with the interest
4. Transfer the pool shares to another account to bypass the lock-up period check
5. Withdraw the pool and receive the initial flash-loan amount and 99% of the interest amount

```
function deposit(
        uint256 underlyingTokenAmount,
        string memory version
    ) external checkVersion(version) ifNotPaused {
        lastDepositTimestamp[msg.sender] = block.timestamp;

        uint256 poolBalance = underlyingToken.balanceOf(address(this));
        underlyingToken.safeTransferFrom(msg.sender, address(this), underlyingTokenAmount);
        underlyingTokenAmount = underlyingToken.balanceOf(address(this)) - poolBalance;

        uint256 rTokenAmount = stablecoinToRToken(underlyingTokenAmount);

        poolValue += underlyingTokenAmount;

        _mint(msg.sender, rTokenAmount);

        emit LiquidityDeposited(block.timestamp, msg.sender, underlyingTokenAmount, rTokenAmount);
    }

    function withdraw(
        uint256 rTokenAmount,
        string memory version
    ) external checkVersion(version) ifNotPaused {
        require(
            block.timestamp > (lastDepositTimestamp[msg.sender] + lockupPeriod),
            Errors.POOL_LOCKUP
        );

        uint256 underlyingTokenAmount = rTokenToStablecoin(rTokenAmount);

        poolValue -= underlyingTokenAmount;

        _burn(msg.sender, rTokenAmount);

        underlyingToken.safeTransfer(msg.sender, underlyingTokenAmount);

        emit LiquidityWithdrawn(block.timestamp, msg.sender, underlyingTokenAmount, rTokenAmount);
    }
    function increasePoolValue(uint256 amount) external onlyRole(Roles.POOL_FIXER) {
        require(amount > 0, Errors.ZERO_VALUE);
    uint256 newPoolValue = poolValue + amount;
    poolValue = newPoolValue;

    underlyingToken.safeTransferFrom(msg.sender, address(this), amount);

    emit PoolValueUpdated(msg.sender, poolValue, newPoolValue, block.timestamp);
}
```

RFIN-26. LOAN LIQUIDATION CHECK DOES NOT TAKE INTEREST INTO ACCOUNT

SEVERITY: Critical

PATH: LoanManager.sol:_isDelinquent:L616-641

REMEDIATION: the function _isDelinquent should use getInterest to obtain the loan's interest and use this together with loan.amount to calculate the correct outstanding amount

STATUS: fixed

DESCRIPTION:

The loan manager uses **_isDelinquent** to check whether a loan is liquidatable. It checks the status and the timestamp of loan before checking the loan's health using the current values of the loan amount and collateral.

However, the health check only uses **loan.amount** to calculate whether the loan should be liquidated based on the value against the collateral. This does not take any accrued interest into account, which only gets added to **loan.amount** upon a repay. If the user never repays the loan, the amount would not get updated.

This can be exploited by a user to forcefully create bad debt by taking on a loan with either or all of high LTV, high interest and long durations. The front-end or bot that use the **view** function **isDelinquent** would not see the loan as unhealthy, because the interest has not yet been added.

The user can then call repay with **1 wei** to update the **loan.amount** and cause a very unhealthy loan to suddenly appear, causing a large amount of bad debt upon liquidation.

Similarly, a user could surprise other users by repaying **1 wei** into their loans and suddenly making their loans liquidatable.

```
function _isDelinquent(
    LoanLib.Loan memory loan,
    IERC20MetadataUpgradeable underlyingToken
) internal view returns (bool) {
    if (
        loan.status == LoanLib.Status.DEFAULT_PART ||
        loan.status == LoanLib.Status.DEFAULT_FULL_PAID ||
        loan.status == LoanLib.Status.DEFAULT_FULL_LIQUIDATED
    ) {
        return false;
    }
    // if loan is due to liquidate - it is delinquent
    if (block.timestamp > loan.liquidationDate) {
        return true;
    }
    // for under-collateralized loan delinquency should be checked on price as well
    if (loan.ltv < 100 ether) {
        return
            convert(
                (loan.amount * 100 ether) / (loan.ltv + LIQUIDATION_THRESHOLD),
                underlyingToken,
                loan.frozenCollateralToken
            ) > loan.frozenCollateralAmount;
    }
    return false;
}
```

RFIN-25. LIQUIDATOR USES INCORRECT REDSTONE OFFSET IN MULTI HOP SWAP FUNCTION

SEVERITY: Critical

PATH: Liquidator:swapLastMultihop:L289-321

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

When the liquidation bot calls **swapLastMultihop**, the function will first call **prepareSwap**, which takes a redstone offset. This offset is later used to append the redstone data to the **LoanManager:convert** call.

However, this offset is incorrectly calculated as **path.length * 32 + version.length + 1** from the tokens path, string version and slippage parameters of the function. This does not take into account the function selector and the ABI-encoded lengths of **path** and **version**.

As a result, the function will always revert if the token uses redstone as price oracle because the data won't be decoded correctly.

Furthermore, because the Liquidator can only **liquidate** each element one-by-one backwards from the liquidity array, it will cause the whole process to always revert.

```
function swapLastMultihop(
    address[] memory path,
    uint8 slippage,
    string memory version
) external checkVersion(version) onlyRole(Roles.LIQUIDATION_BOT) {
    (
        uint256 amountIn,
        uint256 amountOutByPrice,
        uint256 liquidityRecordIndex,
        IERC20MetadataUpgradeable collateralToken,
        IERC20MetadataUpgradeable underlyingToken
    ) = prepareSwap((path.length * 32) + bytes(version).length + 1);

    require(path[0] == address(collateralToken), Errors.LIQUIDATOR_INVALID_PATH);
    require(path[path.length - 1] == address(underlyingToken), Errors.LIQUIDATOR_INVALID_PATH);

    //Slippage is percent with decimals 0
    uint256 amountOutMinimum = amountOutByPrice - ((amountOutByPrice * slippage) / 100);

    uint256[] memory amountOut = swapRouter.swapExactTokensForTokens(
        amountIn,
        amountOutMinimum,
        path,
        address(this),
        block.timestamp
    );

    uint256 resultAmount = amountOut[amountOut.length - 1];

    require(resultAmount >= amountOutMinimum, Errors.LIQUIDATOR_MINIMUM_SWAP_FAILED);

    cleanUpSwap(liquidityRecordIndex, amountIn, resultAmount);
}
    function prepareSwap(
    uint256 redStoneOffset
)
    internal
    returns (
        uint256 amountIn,
        uint256 amountOutPriced,
        uint256 liquidityRecordIndex,
        IERC20MetadataUpgradeable collateralToken,
        IERC20MetadataUpgradeable underlyingToken
    )
```

```
{
    require(address(swapRouter) != address(0), Errors.ZERO_ADDRESS);

    require(liquidity.length > 0, Errors.LIQUIDATOR_NOTHING_TO_SWAP);

    liquidityRecordIndex = liquidity.length - 1;
    LiquidityToSwap memory liquidityRecord = liquidity[liquidityRecordIndex];

    amountIn = liquidityRecord.amount;
    collateralToken = liquidityRecord.collateralToken;
    underlyingToken = liquidityRecord.underlyingToken;

    uint256 balance = collateralToken.balanceOf(address(this));

    require(balance >= liquidityRecord.amount, Errors.LIQUIDATOR_INSUFFICIENT_FUNDS);

    collateralToken.approve(address(swapRouter), liquidityRecord.amount);

    (bool success, bytes memory result) = address(loanManager).call(
        abi.encodePacked(
            abi.encodeWithSelector(
                ILoanManager.convert.selector,
                amountIn,
                collateralToken,
                underlyingToken
            ),
            msg.data[redStoneOffset:]
        )
    );

    require(success, Errors.ZERO_VALUE);

    amountOutPriced = abi.decode(result, (uint256));
}
```

Calculate the offset in Liquidator.sol on line 300 correctly using the ABI encoding's size: Contract ABI Specification — Solidity 0.8.22 documentation. For example a call to swapLastMultihop([address(type(uint160).max), address(type(uint160).max)], 1, "1") would be encoded as:

```
0x
8f756fc4
0000000000000000000000000000000000000000000000000000000000000040
00000000000000000000000000000000000000000000000000000000000000a0
0000000000000000000000000000000000000000000000000000000000000002
000000000000000000000000ffffffffffffffffffffffffffffffffffffffff
000000000000000000000000ffffffffffffffffffffffffffffffffffffffff
0000000000000000000000000000000000000000000000000000000000000001
3100000000000000000000000000000000000000000000000000000000000000
```

which has a size of 260 bytes, much more than the calculated 2 * 32 + 1 + 1 = 66 bytes.

RFIN-2. DIA PRICE ORACLE TIMESTAMP NOT CHECKED

SEVERITY: High

PATH: LoanManager.sol:getPrice:L691-704

REMEDIATION: define a max staleness constant variable and check the timestamp return value of the DiaOracle.getvalue in getPrice

STATUS: fixed

DESCRIPTION:

The DIA oracle (EVM Oracle) returns **(price, timestamp)**, but the timestamp is not checked for staleness nor is there any maximum staleness defined in the code.

If the DIA oracle goes offline then prices become outdated. This is especially troublesome with high volatility assets, which impacts collateral value in loans and liquidation.

```
function getPrice(IERC20MetadataUpgradeable asset) internal view returns (uint256 price) {
    address assetAddress = address(asset);
    string memory assetDiaId = diaId[assetAddress];

    if (bytes(assetDiaId).length > 0) {
        require(address(diaOracle) != address(0), Errors.ZERO_ADDRESS);
        (price, ) = diaOracle.getValue(assetDiaId);
        require(price != 0, Errors.ZERO_VALUE);
        return price;
    }

    price = getOracleNumericValueFromTxMsg(redStoneId[assetAddress]);
    require(price != 0, Errors.ZERO_VALUE);
}
```

RFIN-7. MALICOUS USER CAN ADD MORE ERC777 COLLATERAL THAN HE POSSESSES

SEVERITY: High

PATH: CollateralManager.sol:addCollateral:L220-245

REMEDIATION: add reentrancy protection to user-callable functions of CollateralManager.sol

STATUS: fixed

DESCRIPTION:

The collateral manager contract has a function **addCollateral** for the user to deposit any of the allowed collateral tokens to be used in loans.

If the token with a callback on transfer (akin ERC-777) is included as the allowed collateral, a malicious user possessing X amount of collateral can exploit re-entrancy in **CollateralManager.sol:addCollateral()** to add N*X (N>1) amount of collateral.

Vulnerable code:

```
uint256 collateralManagerBalance = token.balanceOf(address(this));
token.safeTransferFrom(user, address(this), amount);
amount = token.balanceOf(address(this)) - collateralManagerBalance;
```

Attack scenario:

1. The attacker uses a malicious smart contract with X amount of collateral token on its balance.
2. The attacker invokes **CollateralManager.sol:addCollateral()** with small amount (1 wei) via the malicious contract.
3. Token invokes a callback from **safeTransferFrom** back to the malicious contract.
4. Steps 2-4 are repeated N-2 times.
5. Malicious contract invokes **CollateralManager.sol:addCollateral()** for the last time with X amount. And executes callback without re-entering **CollateralManager.sol:addCollateral()**.
6. As a result on L243 of **CollateralManager.sol** collateral balance of the malicious contract will be increased by N*X instead of X.

The same vulnerability exists in **Pool.sol:deposit**, however we expect that underlying tokens of pools will be stable tokens and the likelihood will be low. However, for collateral tokens we do expect volatile tokens.

```
function addCollateral(
    address user,
    IERC20MetadataUpgradeable token,
    uint256 amount
) external payable checkFreezerOrUser(user) ifNotPaused {
    require(amount > 0 || msg.value > 0, Errors.ZERO_VALUE);
    require(allowedCollaterals.includes(token), Errors.COLLATERAL_MANAGER_TOKEN_NOT_SUPPORTED);

    // if token is native then wrap it
    if (msg.value > 0) {
        require(address(wrapper) != address(0), Errors.COLLATERAL_MANAGER_WRAPPER_ZERO);
        require(
            address(wrapper) == address(token),
            Errors.COLLATERAL_MANAGER_TOKEN_IS_NOT_WRAPPER
        );
        wrapper.deposit{value: msg.value}();
        amount = msg.value;
    } else {
        uint256 collateralManagerBalance = token.balanceOf(address(this));
        token.safeTransferFrom(user, address(this), amount);
        amount = token.balanceOf(address(this)) - collateralManagerBalance;
    }

    collateralToUserToAmount[token][user] += amount;
    emit CollateralAdded(user, token, amount);
}
```

RFIN-9. USER MIGHT NOT BE ABLE TO CLAIM NATIVE COLLATERAL

SEVERITY: High

PATH: CollateralManager:claimCollateral:L253-275

REMEDIATION: use the native call function instead of send to avoid out of gas errors

STATUS: fixed

DESCRIPTION:

The function **claimCollateral** can be used to withdraw any collateral. The native function **send** is used for any native/wrapped collateral. This function has a limit of **2300 gas**.

If the depositor is a smart contract that contains a **receive** or **fallback** function that costs more than 2300 gas, then it will never be able to **claimCollateral**, therefore their collateral will be forever stuck in the contract as there is no function to transfer ownership of collateral.

```
function claimCollateral(
    address user,
    IERC20MetadataUpgradeable token,
    uint256 amount
)
    external
    checkAmount(amount)
    checkFreezerOrUser(user)
    checkCollateralBalance(token, user, amount)
    ifNotPaused
{
    collateralToUserToAmount[token][user] -= amount;

    // if token is wrapped native - unwrap it
    if (token == IERC20MetadataUpgradeable(address(wrapper))) {
        wrapper.withdraw(amount);
        require(payable(user).send(amount), Errors.COLLATERAL_MANAGER_NATIVE_TRANSFER);
    } else {
        token.safeTransfer(user, amount);
    }

    emit CollateralClaimed(user, token, amount);
}
```

RFIN-24. LIQUIDATION FEE PERCENTAGE CALCULATION IS INCORRECT

SEVERITY: High

PATH: LoanManager:getDelinquencyInfo:L526-608

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The function **getDelinquencyInfo** is used by the Loan Manager inside of **liquidate** to calculate the correct values during liquidation of a loan.

In the function, on lines 556-577, it performs an early return if the frozen collateral amount covers at least the total remaining amount. If it also covers the liquidation fee, it will return on lines 560-566 with the total amount and the standard 5% as liquidation fee.

If it does not or only partially covers the liquidation fee, then it will return on lines 569-576 and recalculate the liquidation fee percentage. However, this calculation is incorrect, because it divides by the frozen collateral amount instead of the total remaining amount.

For example, if the total remaining amount is 100 ETH, then the liquidation fee would be 5 ETH. In the case where the frozen collateral amount is exactly 105 ETH, it should return the full 5% as liquidation percentage. But instead, it would return (105 - 100) * 100 / 105 = 4.76% and so it will always return a lower amount than it should.

LoanManager:liquidate is called by the **Liquidator.sol** contract. This fee percentage is returned to the Liquidator contract and then stored in a mapping.

Later, when the bot calls **Liquidator.sol:swapLastMultiHop**, it will use the liquidation fee percentage to calculate how much of the total amount to send to the pool.

As a result, this amount will always be too little in this scenario and it would result in loss of user yield.

```
function getDelinquencyInfo(
    uint256 loanId
) public view returns (LoanLib.DelinquencyInfo memory) {
    LoanLib.Loan memory loan = _loans[loanId];

    IERC20MetadataUpgradeable loanToken = loan.pool.underlyingToken();

    require(_isDelinquent(loan, loanToken), Errors.LOAN_MANAGER_LOAN_IS_LIQUID);

    //Calculate accrued interest for now
    uint256 interestAccrued = getInterest(loanId, block.timestamp);

    uint256 remainingAmountAsCollateral = convert(
        loan.amount,
        loanToken,
        loan.frozenCollateralToken
    );

    uint256 remainingInterestAsCollateral = convert(
        interestAccrued,
        loanToken,
        loan.frozenCollateralToken
    );
```

```
  uint256 remainingTotal = remainingAmountAsCollateral + remainingInterestAsCollateral;

  //Add liquidation fee to remaining total
  uint256 liquidationFee = (remainingAmountAsCollateral * LIQUIDATION_THRESHOLD) / 100 ether;

  //If frozen collateral amount of the loan covers loan principal + loan interest accrued
  if (loan.frozenCollateralAmount >= remainingTotal) {
    //if frozen collateral covers remain total and fee

    if (loan.frozenCollateralAmount > remainingTotal + liquidationFee) {
      return
        LoanLib.DelinquencyInfo(
          remainingTotal + liquidationFee,
          0,
          int256(interestAccrued),
          LIQUIDATION_THRESHOLD
        );
    }
    //if frozen collateral covers liquidation fee partially
    return
      LoanLib.DelinquencyInfo(
        loan.frozenCollateralAmount,
        0,
        int256(interestAccrued),
        ((loan.frozenCollateralAmount - remainingTotal) * 100 ether) /
          loan.frozenCollateralAmount
      );
  }

  [..]
}
```

In LoanManager.sol on line 575 it should divide by the remainingTotal instead of the frozenCollateralAmount.

For example:

```
//if frozen collateral covers liquidation fee partially
return
  LoanLib.DelinquencyInfo(
    loan.frozenCollateralAmount,
    0,
    int256(interestAccrued),
    ((loan.frozenCollateralAmount - remainingTotal) * 100 ether) /
      remainingTotal
  );
```

RFIN-3. MISSING SANITY CHECK ON LTV SETTING

SEVERITY: Medium

PATH: SettingsProvider.sol:addPoolToScoreLtvs:L242-250

REMEDIATION: the function addPoolToScoreLtvs should do a sanity check on the provided ltvs values to ensure that it cannot be set to a very large value

STATUS: fixed

DESCRIPTION:

It's crucial that the loan to value (LTV) has a value in a specific range. For example, values **100 ether** or **80 ether** are safe. It's assumed that LTV has value **n * 10**18.**

If the **ltv** value is too big, then **collateralToFreeze** will be zero and a borrower can borrow **amount** of underlying pool tokens for free. On the other hand, if the **ltv** value is small, **collateralToFreeze** is big and it's not possible to borrow. **LoanManager.sol:borrow()** L263-L267:

```
uint256 collateralToFreeze = convert(
    (amount * 100 ether) / ltv,
    vars.underlyingToken,
    collateral
);
```

Method **SettingsProvider.sol:addPoolToScoreLtvs()** that sets ltv value for the pool doesn't check the correctness of the value, it allows any value in **uint256** range for any score.

It’s easy to mistakenly set **ltv** out of the safe range or to set a wrong **ltv** because there is no validation.

```
function addPoolToScoreLtvs(
        IPool pool,
        uint16 score,
        uint256[] memory ltvs
    ) external onlyRole(Roles.ADMIN) {
        require(poolScores[pool].includes[score], Errors.SETTINGS_PROVIDER_SCORE_NOT_SET);
        poolToScoreLtvs[pool][score].addList(ltvs);
        emit PoolToScoreLtvsAdded(msg.sender, pool, score, ltvs, block.timestamp);
    }
```

RFIN-11. LIQUIDATION BOT CAN STEAL COLLATERAL TOKENS FROM LIQUIDATOR

SEVERITY: Medium

PATH: Liquidator.sol

REMEDIATION: define a storage variable acting as settings for a maximum amount of slippage that the liquidation bot can use. This variable should be configurable only by the admin

STATUS: fixed

DESCRIPTION:

The entity with the role **LIQUIDATION_BOT** has a possibility to steal all collateral tokens from the **Liquidator.sol** contract using a sandwich attack.

This is because the slippage value isn't checked inside the function and a safe range isn't enforced by the admin.

When calling **swapLastMultihop()** method it's possible to pass **100** value for the **slippage** parameter, causing **amountOutMinimum** to become 0. Voltage is also a UniswapV2 fork, so it would be trivial to manipulate the pool price using a flash-loan.

```
function swapLastMultihop(
        address[] memory path,
        uint8 slippage,
        string memory version
    ) external checkVersion(version) onlyRole(Roles.LIQUIDATION_BOT) {
        (
            uint256 amountIn,
            uint256 amountOutByPrice,
            uint256 liquidityRecordIndex,
            IERC20MetadataUpgradeable collateralToken,
            IERC20MetadataUpgradeable underlyingToken
        ) = prepareSwap((path.length * 32) + bytes(version).length + 1);

        require(path[0] == address(collateralToken), Errors.LIQUIDATOR_INVALID_PATH);
        require(path[path.length - 1] == address(underlyingToken), Errors.LIQUIDATOR_INVALID_PATH);

        //Slippage is percent with decimals 0
        uint256 amountOutMinimum = amountOutByPrice - ((amountOutByPrice * slippage) / 100);

        uint256[] memory amountOut = swapRouter.swapExactTokensForTokens(
            amountIn,
            amountOutMinimum,
            path,
            address(this),
            block.timestamp
        );

        uint256 resultAmount = amountOut[amountOut.length - 1];

        require(resultAmount >= amountOutMinimum, Errors.LIQUIDATOR_MINIMUM_SWAP_FAILED);

        cleanUpSwap(liquidityRecordIndex, amountIn, resultAmount);
    }
```

RFIN-14. FLAWED ACCOUNTING OF POOL VALUE WHEN UNDERLYING TAKES FEES ON TRANSFER

SEVERITY: Medium

PATH: LoanManager.sol:repay:L336-406

REMEDIATION: update pool balance with the real value which has been received by the pool

STATUS: fixed

DESCRIPTION:

The underlying token of a pool (usually a stable coin) might take fees on transfer. As a result, pool value is updated incorrectly inside of the **LoanManager.sol:repay()** function.

First the underlying token is transferred from **msg.sender** (L385 of **LoanManager.sol**).

```
underlyingToken.safeTransferFrom(msg.sender, address(loan.pool), amount - treasuryShare);
```

Next pool value is updated as follows (L391-L395 of **LoanManager.sol**):

```
    loan.pool.updatePoolValue(
      loan.status == LoanLib.Status.DEFAULT_PART
        ? int256(amount - treasuryShare)
        : int256(interestAccrued - treasuryShare)
    );
```

Eventually, if the underlying token takes fees on transfer pool, it will receive an amount less than accounted for (**amount - treasuryShare**).

RFIN-15. CENTRALISATION RISKS

SEVERITY: Medium

PATH: see description

REMEDIATION: at least use a multi-sig and/or Timelock contract as admin to protect both the users and the protocol from potential losses

STATUS: fixed

DESCRIPTION:

A compromised admin address could atomically change many setter functions in LoanManager, Liquidator, CollateralManager and LimitManager and easily steal funds from the users and the protocol.

For example, in Liquidator.sol, they could set their own address in setLoanManager and then approve all tokens using approveAsset.

```
function setLoanManager(ILoanManager _loanManager) external onlyRole(Roles.ADMIN) {
    require(address(_loanManager) != address(0), Errors.ZERO_ADDRESS);

    emit LoanManagerChanged(msg.sender, loanManager, _loanManager, block.timestamp);

    loanManager = _loanManager;
}
```

```
function setSettingsProvider(
    ISettingsProvider _settingsProvider
) external onlyRole(Roles.ADMIN) {
    settingsProvider = _settingsProvider;
}

function setLimitManager(ILimitManager _limitManager) external onlyRole(Roles.ADMIN) {
    limitManager = _limitManager;
}

function setDiaOracle(address oracle) external onlyRole(Roles.ADMIN) {
    diaOracle = IDIAOracleV2(oracle);
}

function setDiaId(address asset, string memory id) external onlyRole(Roles.ADMIN) {
    diaId[asset] = id;
}

function setRedStoneId(address asset, bytes32 id) external onlyRole(Roles.ADMIN) {
    redStoneId[asset] = id;
}
```

RFIN-10. REMOVING AN ENTRY FROM THE LISTMAP MIGHT RUN OUT OF GAS

SEVERITY: Low

PATH: ListMap.sol:remove:L89-152

REMEDIATION: the current implementation is not optimised. We would recommend to reimplement the ListMap and keep track of an entry's index instead of just a boolean mapping includes

existence of the entry can then still be checked by keeping the 0 index as a special value and checking that the index is then non-zero

removing can then be done in O(1) constant time by swapping the last index with the entry's index, popping the ListMap's array and then clearing the entry's index

STATUS: fixed

DESCRIPTION:

The **ListMap** data structure is used in CollateralManager and SettingsProvider to keep track of allowed collateral tokens and pool settings.

Removing an entry from the **ListMap** using **remove** entails looping through the whole list. Whilst adding an entry to the **ListMap.sol** doesn't require loops.

Therefore there is a chance that a large **ListMap** is created that would result in an out of gas error when deleting an entry.

```
function remove(_uint256 storage listMap, uint256 value) internal {
    for (uint256 i; i < listMap.list.length; i++) {
        if (listMap.list[i] == value) {
            listMap.list[i] = listMap.list[listMap.list.length - 1];
            listMap.list.pop();
            listMap.includes[value] = false;
            return;
        }
    }
    revert(Errors.NO_ELEMENT_IN_ARRAY);
}
```

```
function add(_uint256 storage listMap, uint256 value) internal {
    require(!listMap.includes[value], Errors.ELEMENT_IN_ARRAY);
    listMap.includes[value] = true;
    listMap.list.push(value);
}
```

RFIN-16. LIQUIDATOR SWAP USES BLOCK.TIMESTAMP AS DEADLINE

SEVERITY: Low

PATH: Liquidator.sol:swapLastMultihop:L289-321

REMEDIATION: allow the caller of swapLastMultihop() to choose the deadline parameter

STATUS: fixed

DESCRIPTION:

The liquidation bot will call **swapLastMultihop** after completion of a loan liquidation to swap the tokens and send the liquidation fee to the pool.

Inside the function **swapLastMultihop**, **block.timestamp** is used as the deadline for the swap (L313). This is not effective protection and is the same as setting no deadline at all.

This is because the CL validator can still hold the transaction for any time and execute it whenever, where it would eventually put the **block.timestamp** as deadline and execute successfully.

```
function swapLastMultihop(
    address[] memory path,
    uint8 slippage,
    string memory version
) external checkVersion(version) onlyRole(Roles.LIQUIDATION_BOT) {
    ...
    uint256[] memory amountOut = swapRouter.swapExactTokensForTokens(
        amountIn,
        amountOutMinimum,
        path,
        address(this),
        block.timestamp
    );
    ...
}
```

RFIN-18. CUSTOM ERRORS

SEVERITY: Low

PATH: see description

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

In each contract the validation checks are performed using the **require** function with a reason string.

```
require(s.interestSettings.limit >= amount, Errors.LOAN_MANAGER_LOAN_PARAMS_LIMIT);
```

We would recommend to replace these with custom errors. This should be done by flipping the check.

For example:

```
require(X == Y, "X is not Y");
```

becomes

```
error XnotY(uint, uint);


if (X != Y)
  revert XnotY(X, Y);
```

The usage of custom errors will save a lot of gas during deployment as well as save on code bytesize of the contract. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

RFIN-19. LIQUIDATOR LIQUIDITYTOSWAP STRUCT LOADING OPTIMISATION

SEVERITY: Low

PATH: Liquidator.sol:cleanUpSwap:L262-279

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In the function **cleanUpSwap**, the latest **LiquidityToSwap** struct is loaded from the storage array into memory. This will load all 5 slots of the struct into memory at once, costing 5 **SLOAD**.

Afterwards, only the **pool** field is read once and afterwards the **feePercent** is read directly from storage, costing another **SLOAD**.

Instead, the **pool** field should also be read directly from storage and the **LiquidtyToSwap** memory struct should be removed. This will save a total of 4 **SLOAD**.

```
function cleanUpSwap(
    uint256 liquidityRecordIndex,
    uint256 amountIn,
    uint256 amountOut
) internal {
    LiquidityToSwap memory record = liquidity[liquidityRecordIndex];

    IPool pool = record.pool;

    uint256 feePercent = liquidity[liquidityRecordIndex].feePercent;

    if (feePercent > 0) {
        pool.increasePoolValue((feePercent * amountOut) / 100 ether);
    }
    liquidity.pop();

    emit LiquiditySwapped(liquidityRecordIndex, amountIn, amountOut);
}
```

RFIN-20. LOANMANAGER LOAN STRUCT LOADING OPTIMISATION

SEVERITY: Low

PATH: LoanManager.sol:getInterest, getDelinquencyInfo, isDelinquent (L173-202, L526-608, L610-614)

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

In each of the functions, the corresponding **Loan** struct is loaded from storage into memory. The **Loan** struct is quite large and covers 12 slots. This means that loading it into memory will cost 12 **SLOAD** each time.

The function **getInterest** only accesses 5 different fields/slots, **getDelinquencyInfo** only 4 and **isDelinquent** even just 1 field/slot. A lot of **SLOAD** are therefore wasted.

Instead, for **getInterest** and **getDelinquencyInfo** where the fields are repeatedly used, they should be loaded either on the stack or as special variable struct into memory, and initialised with the value read directly from storage.

For **isDelinquent**, it should read the field directly from storage.

```
LoanLib.Loan memory loan = _loans[loanId];
```

RFIN-23. LIQUIDATED LOAN WITH FULLY SEIZED COLLATERAL IS NOT CLOSED

SEVERITY: Low

PATH: LoanManager.sol:liquidate:L408-497

REMEDIATION: the check in the branch in LoanManager.sol on line 457 should be done using >= instead of >, such that the loan would be closed in the situation as described

STATUS: fixed

DESCRIPTION:

In the function **getDelinquencyInfo**, if a loan gets liquidated and the frozen collateral amount covers the total remaining amount, but not fully the liquidation fee, then the function returns the full frozen collateral amount as to-be-liquidated and 0 as not covered amount. This is returned on lines 569-576.

During the liquidation, the loan amount would become 0 and it is set to **notCovered**. But the loan is not closed in the limit manager in the branch on lines 457-465. This is because in this scenario, the frozen collateral amount is equal to **toLiquidate** and the check is done using **>**.

As a result, the loan gets stuck and the limit manager won’t decrease the total amount of open loans for the user. The loan can also not be repaid, because the loan amount is 0.

```
function getDelinquencyInfo(
    uint256 loanId
) public view returns (LoanLib.DelinquencyInfo memory) {
    [..]
    if (loan.frozenCollateralAmount >= remainingTotal) {
        //if frozen collateral covers remain total and fee

        if (loan.frozenCollateralAmount > remainingTotal + liquidationFee) {
            return
                LoanLib.DelinquencyInfo(
                    remainingTotal + liquidationFee,
                    0,
                    int256(interestAccrued),
                    LIQUIDATION_THRESHOLD
                );
        }
        //if frozen collateral covers liquidation fee partially
        return
            LoanLib.DelinquencyInfo(
                loan.frozenCollateralAmount,
                0,
                int256(interestAccrued),
                ((loan.frozenCollateralAmount - remainingTotal) * 100 ether) /
                    loan.frozenCollateralAmount
            );
    }
    [..]
}

function liquidate(
    uint256 loanId,
    string memory version
)
    external
    whenNotPaused
    nonReentrant
    checkVersion(version)
    onlyRole(Roles.LIQUIDATOR)
    returns (IERC20MetadataUpgradeable, IERC20MetadataUpgradeable, uint256, IPool, uint256)
{
    //Method offset is 4(selector) + 32 (loanId) + dynamic length of version
    LoanLib.DelinquencyInfo memory info = getDelinquencyInfo(loanId);

    [..]
    loan.amount = info.notCovered;

    [..]

    uint256 unfrozenCollateral;
    //If loan frozen collateral exceeds amount of collateral to liquidate we can unfreeze remaining amount
    if (loan.frozenCollateralAmount > info.toLiquidate) {
        unfrozenCollateral = loan.frozenCollateralAmount - info.toLiquidate;
        collateralManager.unfreeze(
            loan.borrower,
            loan.frozenCollateralToken,
            unfrozenCollateral
        );
        limitManager.onLoanFulfillment(loan.borrower, loan.pool);
    }
    [..]
}
```

RFIN-6. OWNER CAN BLOCK WITHDRAWALS OF POOL AND COLLATERAL TOKENS

SEVERITY: Informational

PATH: Pool.sol, CollateralManager.sol

REMEDIATION: because of the lack of an emergency withdraw, we would like to recommend considering the case where a user can always withdraw their deposited pool tokens and collateral tokens by removing the ifNotPaused modifier on these functions

STATUS: fixed

DESCRIPTION:

There is an **ifNotPaused** modifier in the **withdraw** function (lines 143-161) in the **Pool.sol** contract. As a result, the owner can immediately suspend withdrawals of underlying tokens. It is advisable that the pause flag should only be applied to transformative functions, as it can severely impact users when they are unable to withdraw an already deposited amount.

The same risk is also present in the **claimCollateral** function (lines 253-275) in the **CollateralManager.sol** contract.

```
function withdraw(
    uint256 rTokenAmount,
    string memory version
) external checkVersion(version) ifNotPaused {
    require(
        block.timestamp > (lastDepositTimestamp[msg.sender] + lockupPeriod),
        Errors.POOL_LOCKUP
    );

    uint256 underlyingTokenAmount = rTokenToStablecoin(rTokenAmount);

    poolValue -= underlyingTokenAmount;

    _burn(msg.sender, rTokenAmount);

    underlyingToken.safeTransfer(msg.sender, underlyingTokenAmount);

    emit LiquidityWithdrawn(block.timestamp, msg.sender, underlyingTokenAmount, rTokenAmount);
}
```

```
function claimCollateral(
    address user,
    IERC20MetadataUpgradeable token,
    uint256 amount
)
    external
    checkAmount(amount)
    checkFreezerOrUser(user)
    checkCollateralBalance(token, user, amount)
    ifNotPaused
{
    collateralToUserToAmount[token][user] -= amount;

    // if token is wrapped native - unwrap it
    if (token == IERC20MetadataUpgradeable(address(wrapper))) {
        wrapper.withdraw(amount);
        require(payable(user).send(amount), Errors.COLLATERAL_MANAGER_NATIVE_TRANSFER);
    } else {
        token.safeTransfer(user, amount);
    }

    emit CollateralClaimed(user, token, amount);
}
```

RFIN-8. CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH: LoanManager.sol, Constants.sol, ContractsVersions.sol, Errors.sol, Roles.sol

REMEDIATION: the mentioned variables should be marked as private instead of public

STATUS: fixed

DESCRIPTION:

In mentioned contracts, there are constant variables that are declared **public**. However, 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 value outside of where it's used, and won't add another entry to the method ID table. If necessary, the values can still be read from the verified contract source code:

```
uint256 public constant LIQUIDATION_THRESHOLD = 5 ether;
```

RFIN-21. LIQUIDATOR SET SWAP ROUTER INCORRECT FUNCTION NAME

SEVERITY: Informational

PATH: Liquidator.sol:setUniSwapV3Router:L118-124

REMEDIATION: correct the function name to setVoltageRouter

STATUS: fixed

DESCRIPTION:

The function **setUniSwapV3Router** has an incorrect name because it sets the router for the Voltage protocol instead of UniSwapV3.

```
function setUniSwapV3Router(IVoltageRouter _swapRouter) external onlyRole(Roles.ADMIN) {
    require(address(_swapRouter) != address(0), Errors.ZERO_ADDRESS);

    emit SwapRouterChanged(msg.sender, swapRouter, _swapRouter, block.timestamp);

    swapRouter = _swapRouter;
}
```

RFIN-22. EXTERNAL FUNCTION PARAMETERS SHOULD BE MARKED AS CALLDATA

SEVERITY: Informational

PATH: see description

REMEDIATION: mark the parameters as calldata instead of memory and update the ListMap library to handle calldata as well

STATUS: fixed

DESCRIPTION:

There are various locations in the code where an **external** function has **memory** input parameters. These parameters should be marked as **calldata** in favour of gas savings.

- Liquidator.sol:swapLastMultihop
- SettingsProvider.sol:addPools, removePools, addPoolCollaterals, removePoolCollaterals, addPoolScores, removePoolScores, addPoolToScoreLtvs, removePoolToScoreLtvs, addPoolToScoreDurations, removePoolToScoreDurations, setPoolToScoreToLtvToDuratioToInterestSettings
- CollateralManager.sol:addCollateral, removeCollaterals

```
function swapLastMultihop(
   address[] memory path,
   uint8 slippage,
   string memory version
) external {
  [..]
}
```

hexens