hexens × Pencils Protocol

AUG.24

SECURITY REVIEW REPORT FOR PENCILS PROTOCOL

CONTENTS

- About Hexens
- Executive summary
    - Overview
    - Scope
- Auditing details
- Severity structure
    - Severity characteristics
    - Issue symbolic codes
- Findings summary
- Weaknesses
    - The amount of shares received from deposit can be manipulated by borrowing/repaying
    - The `openPositionV1()` function calls `borrowForPosition()` to the wrong address
    - Missing token approval when calling repayForPosition() from PositionManager contract
    - `openPositionV1` function deposits insufficient tokens into the strategy vault
    - Liquidation doesn't check if the position is closed, allowing an attacker to liquidate a position multiple times and drain all funds
    - Attacker can leverage flashloans to steal rewards from other users
    - Total borrow amount doesn't accrue total interest from all loans, causing unfair utilization and borrowing situations
    - Due to missing approval certain functionality will not work

- The bad debt positions will be unable to liquidated or closed
- Insufficient pausing logic
- An attacker can create a position that cannot be liquidated
- Interest should be accrued before changing the rate
- Read-only reentrancy may be possible in external integrations
- Use custom errors
- Missing pausing logic for addMarginPositionV1() function

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 focuses on the Pencils Protocol, a next-generation decentralized platform that provides auction services for blockchain-native assets and real-world assets (RWAs), along with unified and leveraged yield aggregation services to help users maximize asset utilization. Additionally, Pencils Protocol serves as a native gateway for liquid staking and restaking assets on Scroll.

The audit is confined to the smart contracts located in the `src/farm` and `src/stake` directories of the repository and was completed within a one-week timeframe.

During the audit, we identified six critical vulnerabilities that could potentially be exploited by attackers to steal funds from the contracts. In addition, we discovered three high-severity vulnerabilities, three medium-level issues, and three informational issues.

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

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

SCOPE

The analyzed resources are located on:

https://github.com/PencilsProtocol/audit-pencils-protocol-v2/commit/cbc4c825c17de262d27ec77e1e8e64bcc1b72ec2

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

https://github.com/PencilsProtocol/audit-pencils-protocol-v2/commit/9bfc064814050b37f3cc20f221dbd9ba0917aacc

AUDITING DETAILS

STARTED
05.08.2024

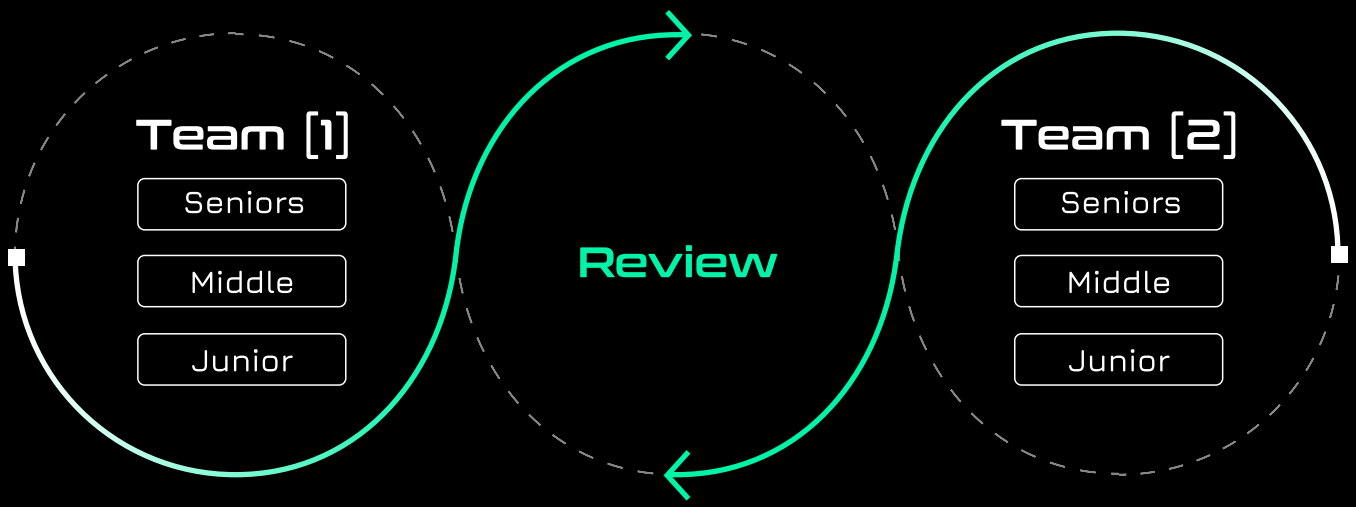
DELIVERED
12.08.2024

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.

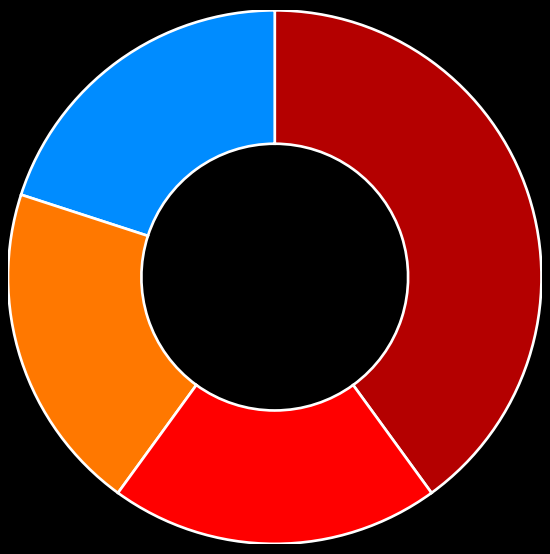
ISSUE SYMBOLIC CODES

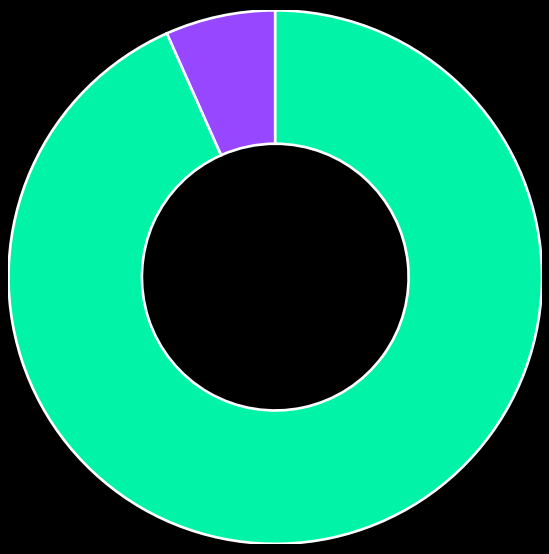
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	6
High	3
Medium	3
Low	0
Informational	3

Total: 15





WEAKNESSES

This section contains the list of discovered weaknesses.

PENCIL-2

THE AMOUNT OF SHARES RECEIVED FROM DEPOSIT CAN BE MANIPULATED BY BORROWING/REPAYING

SEVERITY: Critical

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Vault::deposit() function calculates the received shares by this calculation:

```
share = (amountToken * totalSupply()) / totalToken();
```

While **totalToken()** is base token balance of this contract.

```
function totalToken() public view override returns (uint256) {
    return SafeToken.myBalance(token);
}
```

However, this balance can be increased/decreased by borrowing/repaying, allowing attackers to manipulate the amount of shares received and then steal funds from users.

An example scenario:
1. The first depositor deposits 100 ETH tokens (100e18) and receives 100e18 shares
2. Attacker borrows 99 ETH, causing **totalToken()** to decrease to 1e18
3. Attacker deposits 1 ETH token (1e18) and receives 100e18 shares
4. Attacker repays the loan and redeems 100e18 shares to collect 50 ETH tokens.

```
function deposit(uint256 amountToken) external payable nonReentrant {
    uint256 share = 0;
    uint256 MINIMUM_LIQUIDITY = 10 ** 3;

    // create dead share when first deposit
    if (totalSupply() == 0) {
        require(amountToken > MINIMUM_LIQUIDITY, "MINIMUM_LIQUIDITY");
        share = amountToken - MINIMUM_LIQUIDITY;
        _mint(address(this), 10 ** 3);
    } else {
        // general mint logic
        share = (amountToken * totalSupply()) / totalToken();
    }

    require(share > 0, "insufficient share minted");
    _safeWrap(amountToken);
    _mint(msg.sender, share);
    lenderLayout().totalSupplyAmount += amountToken;
}
```

The **totalToken** function should include **totalBorrowAmount**.

```
function totalToken() public view override returns (uint256) {
    return SafeToken.myBalance(token) + totalBorrowAmount;
}
```

However, to obtain the correct **totalBorrowAmount**, we suggest accruing the total interest of all loans into **totalBorrowAmount** in **_accrueInternal()** function, and first triggering that function for every deposit.

THE `OPENPOSITIONV1()` FUNCTION CALLS `BORROWFORPOSITION()` TO THE WRONG ADDRESS

SEVERITY: Critical

PATH:

src/position/PositionManager.sol#L83

REMEDIATION:

The target address of borrowForPosition call should be _getLendingVault(borrowAsset_) instead of borrowAsset_ .

STATUS: Fixed

DESCRIPTION:

In PositionManager contract, **openPositionV1()** function attempts to borrow from the lending vault by calling the **borrowForPosition()** function. However, it calls **borrowAsset_**, which is the underlying asset of the vault:

```
IVault(borrowAsset_).borrowForPosition(cachedPositionId, borrowAsset_,
borrowAmount_);
```

Calling the wrong address will result in borrowing never succeed. The target address of this call should be **_getLendingVault(borrowAsset_)** instead of **borrowAsset_**.

MISSING TOKEN APPROVAL WHEN CALLING REPAYFORPOSITION() FROM POSITIONMANAGER CONTRACT

SEVERITY: Critical

PATH:

src/position/PositionManager.sol#L188-L191

src/position/PositionManager.sol#L214-L216

src/position/PositionManager.sol#L122-L124

REMEDIATION:

Consider give the token allowance to the pos.lender contract before triggering the repayForPosition() function.

STATUS: Fixed

DESCRIPTION:

The **Vault.repayForPosition()** function is designed for the **PositionManager** contract to repay the debt of a position. This function pulls tokens from the **PositionManager** contract, which requires token approval from the **PositionManager** before it is triggered.

However, within the **PositionManager.closePositionV1()** and **PositionManager.repayPositionV1()** functions, there is no **approve()** call to grant permission to the **Vault** contract. This omission causes the subsequent **Vault.repayForPosition()** invocation to revert.

The impact is severe because users are unable to repay their debt.

```
// pull repayAsset
IERC20(repayAsset).safeTransferFrom(msg.sender, address(this), repayAmount);
// repay for position
lender.repayForPosition(positionId, repayAsset, repayAmount);
```

```
// calculate repay amount and perform repay
uint256 repayAmount = IVault(pos.lender).borrowBalanceOf(positionId);
IVault(pos.lender).repayForPosition(positionId, pos.borrowAsset,
repayAmount);
```


```
// calculate repay amount and perform repay
uint256 repayAmount = IVault(pos.lender).borrowBalanceOf(positionId);
IVault(pos.lender).repayForPosition(positionId, pos.borrowAsset,
repayAmount);
```

`OPENPOSITIONV1` FUNCTION DEPOSITS INSUFFICIENT TOKENS INTO THE STRATEGY VAULT

SEVERITY: Critical

PATH:

src/position/PositionManager.sol#L86

REMEDIATION:

It should deposit balanceActual tokens instead of underlyingAmount_ .

STATUS: Fixed

DESCRIPTION:

openPositionV1() function attempts to pull **underlyingAmount_** amount of tokens from the user and borrow another amount of tokens from the lending vault, then deposit the tokens into the strategy vault for farming. However, the deposit call in this function only uses the **underlyingAmount_** of tokens, resulting in the borrowed tokens not being deposited into the strategy vault, causing loss from interest and positions may be unable to close.

```
uint256 shareAmount_ = IERC4626(strategyVault_).deposit(underlyingAmount_,
address(this));
```

```
function openPositionV1(
        uint256 leverageRatio_,
        address underlyingAsset_,
        uint256 underlyingAmount_,
        address borrowAsset_,
        uint256 borrowAmount_,
        address strategyVault_
    ) external nonReentrant whenNotPaused {
        /**
         * Check
         */
        // 1. check leverage ratio
        if (leverageRatio_ > 3) {
            revert Errors.Position_InvalidLeverageRatio();
        }
        // 2. check underlyingAsset in whitelist
        if (!_containUnderlyingAsset(underlyingAsset_)) {
            revert Errors.UnderlyingAsset_NotInWhiteList(underlyingAsset_);
        }
        // 3. check lending vault for borrowAsset is in whitelist
        if (!_containLendingVault(borrowAsset_)) {
            revert Errors.BorrowAsset_NotInWhiteList(borrowAsset_);
        }
        // 4. underlyingAsset == borrowAsset
        if (!(underlyingAsset_ == borrowAsset_)) {
            revert Errors.Position_V1_NotTheSameToken();
        }
        // 5. check strategy in whitelist
        if (!_containStrategy(strategyVault_)) {
            revert Errors.StrategyAsset_NotInWhiteList(strategyVault_);
        }
        // 5. check underlying vaule
        uint256 balanceActual = underlyingAmount_ + borrowAmount_;
        if ((underlyingAmount_ * leverageRatio_) < balanceActual) {
            revert Errors.Position_BorrowTooMuch();
        }
```

```
        /**
         * Effect & Interaction
         */
        // cached position id
        uint256 cachedPositionId = _getPositionId();
        // pull underlyingAsset
        IERC20(underlyingAsset_).safeTransferFrom(msg.sender, address(this),
underlyingAmount_);
        // pull borrowAsset
        IVault(borrowAsset_).borrowForPosition(cachedPositionId,
borrowAsset_, borrowAmount_);
        // deposit to strategy
        IERC20(underlyingAsset_).approve(strategyVault_, balanceActual);
        uint256 shareAmount_ =
IERC4626(strategyVault_).deposit(underlyingAmount_, address(this));
        // open position
        _openPosition(
            leverageRatio_,
            msg.sender,
            underlyingAsset_,
            underlyingAmount_,
            _getLendingVault(borrowAsset_),
            borrowAsset_,
            borrowAmount_,
            strategyVault_,
            shareAmount_
        );
        // emit event
        emit Events.OpenPosition(msg.sender, cachedPositionId);
    }
```

PENCIL-12

LIQUIDATION DOESN'T CHECK IF THE POSITION IS CLOSED, ALLOWING AN ATTACKER TO LIQUIDATE A POSITION MULTIPLE TIMES AND DRAIN ALL FUNDS

SEVERITY: Critical

PATH:

src/position/PositionManager.sol#L198-L249

REMEDIATION:

liquidatePositionV1 function should include the following check:

```
if (pos.state != StateCode.RUNNING) {
    revert Errors.Position_NotRunning();
}
```

STATUS: Fixed

DESCRIPTION:

After liquidation or closure, only the position state will be set to **CLOSE**. However, during the liquidation process of the **liquidatePositionV1()** function, it doesn't check if the position is closed, allowing a closed position to be liquidated.

Therefore, an attacker can liquidate an unhealthy position multiple times, which will redeem unlimited strategy shares of the **PositionManager** and drain all funds in the **PositionManager** contract.

```
function liquidatePositionV1(uint256 positionId) external
onlyRole(LIQUIDATOR) nonReentrant {
        /**
         * check
         */
        if (!isLiquidatableV1(positionId)) {
            revert Errors.Position_V1_NotLiquidatable(positionId);
        }

        /**
         * effect and interaction
         */
        GlobalPositionStruct storage $ = positionLayout();
        SinglePositionStruct storage pos = $.positions[positionId];

        // redeem from strategy vault
        uint256 redeemAmount =
IERC4626(pos.strategy).redeem(pos.strategyAmount, address(this),
address(this));
        // calculate repay amount and perform repay
        uint256 repayAmount =
IVault(pos.lender).borrowBalanceOf(positionId);
        IVault(pos.lender).repayForPosition(positionId, pos.borrowAsset,
repayAmount);
        // send back remain part to liquidator
        IERC20(pos.underlyingAsset).safeTransfer(msg.sender, redeemAmount -
repayAmount);
        // close position
        _closePosition(positionId);
        // emit event
        emit Events.Liquidated(pos.farmer, positionId);
    }

    function isLiquidatableV1(uint256 positionId) public view returns (bool)
{
        GlobalPositionStruct storage $ = positionLayout();
        SinglePositionStruct storage pos = $.positions[positionId];
```

```
        // underwater amount
        uint256 underwaterAmount =
            pos.underlyingAmount * _getUnderwaterRatio(pos.underlyingAsset)
/ UNDERWATER_DENOMINATOR;
        // actual amount
        uint256 actualAmount;
        {
            // value reflection from borrow
            uint256 borrowAmount =
IVault(pos.lender).borrowBalanceOf(positionId);
            uint256 borrowInterestAmount = borrowAmount > pos.borrowAmount ?
borrowAmount - pos.borrowAmount : 0;
            // value reflection from share
            uint256 shareValue =
IERC4626(pos.strategy).convertToAssets(pos.strategyAmount);
            (bool isNeg, uint256 shareInterestValue) = shareValue >
pos.underlyingAmount + pos.borrowAmount
                ? (false, shareValue - pos.underlyingAmount -
pos.borrowAmount)
                : (true, pos.underlyingAmount + pos.borrowAmount -
shareValue);
            // calc actualAmount
            actualAmount = pos.underlyingAmount - borrowInterestAmount;
            actualAmount = isNeg ? actualAmount - shareInterestValue :
actualAmount + shareInterestValue;
        }

        return actualAmount < underwaterAmount;
    }
```

PENCIL-15

ATTACKER CAN LEVERAGE FLASHLOANS TO STEAL REWARDS FROM OTHER USERS

SEVERITY: Critical

REMEDIATION:

Need to update storedTotalAssets before minting new shares.

STATUS: Fixed

DESCRIPTION:

The problem occurs when new shares are minted before the rewards are realized and the **storedTotalAssets** amount is updated in the **_strategyAdd** function. This order leads to an incorrect reward calculation because the asset balance is not properly updated before the new shares are added.

Consider the following scenario:

Initial Setup:

- Alice opens position for 100 USDC.
- Bob opens position for 900 USDC.
- The total in Compound is 1000 USDC.

After One Week:

- The Compound market earns 10% interest, making the total 1100 USDC.
- Alice and Bob's rewards haven't been realized, so the strategy's storedTotalAsset is still 1000 USDC.

Flash Loan Exploit:

- The attacker flash loans 10,000 USDC and deposits it into the strategy.
- The strategy's storedTotalAsset updates to 11,000 USDC (10,000 USDC flash loan + 1000 USDC).
- The strategy mints shares for the attacker based on the outdated storedTotalAsset, not accounting for the 100 USDC interest earned.
- Calls the **redeem** function on the strategy to trigger **_strategyRemove** and realize rewards based on the outdated **storedTotalAsset**, which dilutes the shares of existing users.

Impact:

- Before the flash loan, Alice had 10% ownership of the 1100 USDC rewards
- After the flash loan, her effective ownership is diluted to just 1% of the 11,100 USDC.
- The attacker can withdraw the rewards belonging to Bob and Alice, leaving them with far less than they should have received.

TOTAL BORROW AMOUNT DOESN'T ACCRUE TOTAL INTEREST FROM ALL LOANS, CAUSING UNFAIR UTILIZATION AND BORROWING SITUATIONS

SEVERITY: High

PATH:

src/farm/Vault.sol#L264
src/farm/Vault.sol#L312

REMEDIATION:

The totalBorrowAmount should accrue the total interest from all loans during every borrowing/repaying action via the accrueInternal function, and it shouldn't accrue the interest of each position afterward. The calculation can be as follows:

$.totalBorrowAmount += safe64(mulFactor($.totalBorrowAmount, borrowRate * timeElapsed));

STATUS: Fixed

DESCRIPTION:

In the Vault contract, the functions **borrowForPosition** and **repayForPosition** only accrue the updated interest of that position to **lenderLayout().totalBorrowAmount**. Therefore, the frequency and number of these actions affect **totalBorrowAmount** differently, making the utilization and interest rate of borrowing unstable. This creates an unfair and manipulatable situation where an action of borrowing or repaying can increase the interest of the next loan.

Example:

1. A creates the first borrowing position in the vault with a principal value of 1000 WETH, and the current total supply is 2000 WETH. After a long period, the interest on A's loan amounts to 200 WETH, but it has not accrued because A hasn't repaid or borrowed again
2. B creates another position, borrowing 100 WETH and repaying it after 1 month. There are 2 possible scenarios:
   a. A hasn't taken any action to accrue interest. In this case, the interest on B's loan will be calculated based on the current utilization rate of 50% (1000 WETH total borrowed / 2000 WETH total supply)
   b. A decided to repay 1 wei for their position, causing the interest on A's loan to be accrued to the total borrow. Consequently, the interest on B's loan will be calculated based on the updated utilization rate of 60% (1200 WETH total borrowed / 2000 WETH total supply).

```
function borrowForPosition(uint256 positionId, address targetToken, uint256
amountToBorrow)
    public
    nonReentrant
    onlyRole(POSITION_MANAGER)
{
    // check target is base token in vault
    require(targetToken == token, "invalid base token");

    // update the rate
    accrueInternal();

    // position accounting
    PositionStruct storage $position = positionLayout();
    PositionBasic storage srcUser = $position.basic[positionId];

    // balance & principal before
    uint256 srcBalance = srcUser.borrowAmount;
    int256 srcPrincipal = srcUser.principal;
    // balance after
    int256 srcBalanceWithInterest = presentValue(srcPrincipal);
    int256 srcBalanceNew = srcBalanceWithInterest -
signed256(amountToBorrow);
```

```
    // principal after
    int256 srcPrincipalNew = principalValue(srcBalanceNew);
    // srcPrincipalNew should not possitive due to the Vault only provide
borrow/repay here
    require(srcPrincipalNew <= 0, "possitive");

    /**
      * effect & interaction
      */
    // update srcUser after
    srcUser.borrowAmount = uint256(-srcBalanceNew);
    srcUser.principal = srcPrincipalNew;
    // update totalBorrowAmount
    // totalBorrowAmount += interest + amountToBorrow
    //                   += ((alreadyBorrow + interest) - alreadyBorrow +
amountToBorrow)
    lenderLayout().totalBorrowAmount += (uint256(-srcBalanceWithInterest) -
srcBalance + amountToBorrow);

    // doTransferOut
    targetToken.safeTransfer(msg.sender, amountToBorrow);

    // emit event
    emit Borrow(msg.sender, positionId, amountToBorrow);
}
```

DUE TO MISSING APPROVAL CERTAIN FUNCTIONALITY WILL NOT WORK

SEVERITY: High

PATH:

src/position/PositionManager.sol#L157-L159

REMEDIATION:

Give the token allowance to the pos.strategy contract before triggering the IERC4626(pos.strategy).deposit() function.

STATUS: Fixed

DESCRIPTION:

In the **PositionManager** contract, the **addMarginPositionV1()** function is used to increase the principal value through deposits. However, this function does not handle token approval (for strategy), which causes the transaction to revert.

```
IERC20(underlyingAsset).safeTransferFrom(msg.sender, address(this),
underlyingAmount);
// deposit to strategy
uint256 shareAmount = IERC4626(pos.strategy).deposit(underlyingAmount,
address(this));
```

```
function deposit(uint256 underlyingAmount, address receiver) public virtual
returns (uint256) {
        if (underlyingAmount > _maxDeposit(receiver)) {
            // require(underlyingAmount <= _maxDeposit(receiver));
            revert ERC4626_DepositThanMax();
        }

        uint256 shareAmount = _previewDeposit(underlyingAmount);
        _deposit(msg.sender, receiver, underlyingAmount, shareAmount);

        return shareAmount;
    }
```

```
function _deposit(address caller, address receiver, uint256
underlyingAmount, uint256 shareAmount)
        internal
        virtual
    {
        ERC4626BaseStorage.ERC4626BaseLayout storage $ =
ERC4626BaseStorage.layout();

        // slither-disable-next-line reentrancy-no-eth
        SafeERC20.safeTransferFrom(IERC20($.underlyingAsset), caller,
address(this), underlyingAmount);
        _mint(receiver, shareAmount);

        // emit event
        emit Deposit(caller, receiver, underlyingAmount, shareAmount);
    }
```

PENCIL-14

THE BAD DEBT POSITIONS WILL BE UNABLE TO LIQUIDATED OR CLOSED

SEVERITY: High

PATH:

src/position/PositionManager.sol#L213-L218

REMEDIATION:

liquidatePositionV1() function should consider the case when redeemAmount < repayAmount (position incur bad debt). In this case, it should repay all of the redeemAmount for vault and close the position.

STATUS: Fixed

DESCRIPTION:

When a position is closed or liquidated, it will transfer the difference between the tokens redeemed and the tokens needed to repay. However, if the strategy is in a pessimistic situation for a long time, causing losses greater than the underlying tokens, the amount received from redemption will be smaller than the amount needed to repay.
In this case, the position will incur a bad debt for the **PositionManager** and will be unable to be closed or liquidated due to the underflow subtraction (**redeemAmount - repayAmount**).

The vault is at risk of never being repaid for the bad debt positions, affecting the liquidity for vault lenders.

```
// redeem from strategy vault
uint256 redeemAmount = IERC4626(pos.strategy).redeem(pos.strategyAmount,
address(this), address(this));
// calculate repay amount and perform repay
uint256 repayAmount = IVault(pos.lender).borrowBalanceOf(positionId);
IVault(pos.lender).repayForPosition(positionId, pos.borrowAsset,
repayAmount);
// send back remain part to liquidator
IERC20(pos.underlyingAsset).safeTransfer(msg.sender, redeemAmount -
repayAmount);
```

```
function liquidatePositionV1(uint256 positionId) external
onlyRole(LIQUIDATOR) nonReentrant {
    /**
      * check
      */
    if (!isLiquidatableV1(positionId)) {
        revert Errors.Position_V1_NotLiquidatable(positionId);
    }

    /**
      * effect and interaction
      */
    GlobalPositionStruct storage $ = positionLayout();
    SinglePositionStruct storage pos = $.positions[positionId];

    // redeem from strategy vault
    uint256 redeemAmount = IERC4626(pos.strategy).redeem(pos.strategyAmount,
address(this), address(this));
    // calculate repay amount and perform repay
    uint256 repayAmount = IVault(pos.lender).borrowBalanceOf(positionId);
    IVault(pos.lender).repayForPosition(positionId, pos.borrowAsset,
repayAmount);
    // send back remain part to liquidator
    IERC20(pos.underlyingAsset).safeTransfer(msg.sender, redeemAmount -
repayAmount);
    // close position
    _closePosition(positionId);
    // emit event
    emit Events.Liquidated(pos.farmer, positionId);
}
```

INSUFFICIENT PAUSING LOGIC

SEVERITY: Medium

PATH:

src/farm/Vault.sol#L366-L372
src/farm/VaultEth.sol#L373-L379

REMEDIATION:

Consider adding a whenNotPaused modifier where it is necessary.

STATUS: Fixed

DESCRIPTION:

The **Vault** and **VaultEth** contracts implement pausing logic from **PauseableExternal** and implements **pause** and **unpause** to set the **paused** boolean variable. However, the contract does not use any function-blocking modifiers, such as **whenNotPaused**.

This could lead to a scenario where the **PAUSER** pauses the protocol, yet users are still able to call functions and potentially lose their funds in case of an emergency.

```
function setPause() public override whenNotPaused onlyRole(PAUSEABLE) {
        super.setPause();
    }
```

```
function setUnpause() public override whenPaused onlyRole(PAUSEABLE) {
        super.setUnpause();
    }
```

PENCIL-16

AN ATTACKER CAN CREATE A POSITION THAT CANNOT BE LIQUIDATED

SEVERITY: Medium

PATH:

src/position/PositionManager.sol#L229-L231

REMEDIATION:

Consider modifying line 248 of the contract PositionManager to:

```
- return actualAmount < underwaterAmount;
+ return actualAmount <= underwaterAmount;
```

STATUS: Fixed

DESCRIPTION:

The function **PositionManager.isLiquidatableV1()** determines whether a position is eligible for liquidation by comparing the **actualAmount** with the **underwaterAmount**. The **underwaterAmount** is calculated using the formula: **pos.underlyingAmount * _getUnderwaterRatio(pos.underlyingAsset) / UNDERWATER_DENOMINATOR**.

This formula involves a round-down calculation, which means that if the **pos.underlyingAmount** is small enough, the **underwaterAmount** could be **0**. Since **0** is the smallest possible value for a **uint256**, the **actualAmount** will always be greater than or equal to the **underwaterAmount**, causing the **isLiquidatableV1()** function to always return **false**.

Based on this analysis, an attacker can create a position with a **pos.underlyingAmount** small enough to result in an **underwaterAmount** of **0**, rendering the position unliquidatable.

```
// underwater amount
uint256 underwaterAmount =
    pos.underlyingAmount * _getUnderwaterRatio(pos.underlyingAsset) /
UNDERWATER_DENOMINATOR;
```

PENCIL-17

INTEREST SHOULD BE ACCRUED BEFORE CHANGING THE RATE

SEVERITY: Medium

PATH:

src/farm/Vault.sol#L352-L360

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **Vault.rateSet()** function updates the interest rate configuration. However, if the interest hasn't been accrued before applying the new configuration, the updated rate takes effect immediately, which can lead to unexpected outcomes (e.g., users being charged higher interest).

```
function rateSet(uint256 kink, uint256 low, uint256 high, uint256 base)
external onlyRole(CONFIG_SETTER) {
    LenderStruct storage $lender = lenderLayout();


    // setter
    $lender.borrowKink = kink;
    $lender.borrowPerSecondInterestRateSlopeLow = low;
    $lender.borrowPerSecondInterestRateSlopeHigh = high;
    $lender.borrowPerSecondInterestRateBase = base;
}
```

Consider accruing the interest before updating the rate.

```
function rateSet(uint256 kink, uint256 low, uint256 high, uint256 base)
external onlyRole(CONFIG_SETTER) {
    LenderStruct storage $lender = lenderLayout();

+    accrueInternal();

    // setter
    $lender.borrowKink = kink;
    $lender.borrowPerSecondInterestRateSlopeLow = low;
    $lender.borrowPerSecondInterestRateSlopeHigh = high;
    $lender.borrowPerSecondInterestRateBase = base;
}
```

READ-ONLY REENTRANCY MAY BE POSSIBLE IN EXTERNAL INTEGRATIONS

SEVERITY: Informational

PATH:

src/farm/Vault.sol#L196-L201
src/farm/VaultEth.sol#L204-L208

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Although the **Vault** and **VaultEth** contracts utilize reentrancy guards, it can still be possible to use the reentrancy in **withdraw()** in case of an external integration of the protocol because the function does not follow the CEI pattern and calls **msg.sender** before the **lenderLayout().totalSupplyAmount -= amount**; storage change.

The reentrancy allows to manipulate **totalSupplyAmount** used in the utilization rate calculations, which affects the **getUtilization()** and **borrowBalanceOf()** public functions.

```
function withdraw(uint256 share) external nonReentrant {
    uint256 amount = share * (totalToken()) / (totalSupply());
    _burn(msg.sender, share);
    _safeUnwrap(msg.sender, amount);
    lenderLayout().totalSupplyAmount -= amount;
}
```

We recommend changing the order of execution, swapping the storage change and the call in places:

```
function withdraw(uint256 share) external nonReentrant {
    uint256 amount = share * (totalToken()) / (totalSupply());
    _burn(msg.sender, share);
    _safeUnwrap(msg.sender, amount);
    lenderLayout().totalSupplyAmount -= amount;
}
```

USE CUSTOM ERRORS

SEVERITY: Informational

STATUS: Acknowledged

DESCRIPTION:

Custom Errors, available from Solidity compiler version 0.8.4, provide benefits such as smaller contract size, improved gas efficiency, and better protocol interoperability. Replace require statements with Custom Errors for a more streamlined and user-friendly experience. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

For example:

```
require(X == Y, "reason");
```

becomes

```
error XnotY(uint, uint);

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

MISSING PAUSING LOGIC FOR ADDMARGINPOSITIONV1() FUNCTION

SEVERITY: Informational

PATH:

src/position/PositionManager.sol#L133-L165

REMEDIATION:

Consider adding a whenNotPaused modifier in the addMarginPositionV1() function.

STATUS: Fixed

DESCRIPTION:

The **PositionManager** contract implements pausing logic from OpenZeppelin's PausableUpgradeable and uses the **whenNotPaused** modifier to restrict function execution when the contract is paused. However, the **addMarginPositionV1()** function lacks this modifier, which is logically inconsistent since other functions, such as **openPositionV1()**, include this modifier.

This inconsistency could lead to a scenario where the **PAUSER** pauses the protocol, yet users can still call **addMarginPositionV1()** and update the supply.

```
function addMarginPositionV1(uint256 positionId, address underlyingAsset,
uint256 underlyingAmount)
        external
        nonReentrant
    {
        GlobalPositionStruct storage $ = positionLayout();
        SinglePositionStruct storage pos = $.positions[positionId];

        /**
         * check
         */
        if (pos.state != StateCode.RUNNING) {
            revert Errors.Position_NotRunning();
        }
        if (pos.farmer != msg.sender) {
            revert Errors.Position_NotFarmer(msg.sender);
        }
        if (pos.underlyingAsset != underlyingAsset) {
            revert Errors.Position_RepayNotUnderlying(underlyingAsset);
        }

        /**
         * effect and interaction
         */
        // pull underlyingAsset
        IERC20(underlyingAsset).safeTransferFrom(msg.sender, address(this),
underlyingAmount);
        // deposit to strategy
        uint256 shareAmount =
IERC4626(pos.strategy).deposit(underlyingAmount, address(this));
        // account for position
        pos.underlyingAmount += underlyingAmount;
        pos.strategyAmount += shareAmount;
        // emit event
        emit Events.AddMargin(pos.farmer, positionId, pos.underlyingAsset,
underlyingAmount);
    }
```


hexens
×
Pencils Protocol