SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Taurus
Prepared by:	Sherlock
Lead Security Expert:	0x52
Dates Audited:	March 6 - March 13, 2023
Prepared on:	May 2, 2023

Introduction

Productive assets is what this cycle is about. Borrow, leverage and earn real yield, the choice is yours. You're going to like what we've built on Arbitrum.

Scope

Repository: protokol/taurus-contracts

Branch: main

Commit: 3759a646f5738890198eb7ae3964e4ecbe952d17

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
6	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

roguereddwarf
cducrest-brainbot
GimelSec
Ruhum
shaka
KingNFT
J4de
cryptostellar5
spyrosonic10
duc
mstpr-brainbot
bytes032

jonatascm
RaymondFam
ck
Bauer
peanuts
imare
Bahurum
yixxas
0x52
nobody2018
SunSec
8olidity
LethL
tvdung94
chaduke
y1cunhui
HonorLt
Chinmay

Issue H-1: Protocol assumes 18 decimals collateral

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/35

Found by

0x52, Bahurum, Bauer, GimelSec, RaymondFam, bytes032, cducrest-brainbot, ck, duc, imare, jonatascm, mstpr-brainbot, peanuts, roguereddwarf, yixxas

Summary

Multiple calculation are done with the amount of collateral token that assume the collateral token has 18 decimals. Currently the only handled collateral (staked GLP) uses 18 decimals. However, if other tokens are added in the future, the protocol may be broken.

Vulnerability Detail

TauMath.sol calculates the collateral ratio (coll * price / debt) as such:

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Libs/TauMath.sol#L18

It accounts for the price decimals, and the debt decimals (TAU is 18 decimals) by multiplying by Constants.precision (`1e18`) and dividing by `10**priceDecimals`. The result is a number with decimal precision corresponding to the decimals of `_coll`.

This `collRatio` is later used and compared to values such as `MIN_COL_RATIO`, `MAX_LIQ_COLL_RATIO`, `LIQUIDATION_SURCHARGE`, or `MAX_LIQ_DISCOUNT` which are all expressed in `1e18`.

Secondly, in `TauDripFeed` the extra reward is calculated as: https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/TauDripFeed.sol#L91

This once again assumes 18 decimals for the collateral. This error is cancelled out when the vault calculates the user reward with: https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L90

However, if the number of decimals for the collateral is higher than 18 decimals, the rounding of `_extraRewardPerCollateral` may bring the rewards for users to 0.

For example: `_tokensToDisburse = 100 e18`, `_currentCollateral = 1_000_000 e33`, then `_extraRewardPerCollateral = 0` and no reward will be distributed.

Impact

Collateral ratio cannot be properly computed (or used) if the collateral token does not use 18 decimals. If it uses more decimals, users will appear way more collateralised than they are. If it uses less decimals, users will appear way less collateralised. The result is that users will be able to withdraw way more TAU than normally able or they will be in a liquidatable position before they should.

It may be impossible to distribute rewards if collateral token uses more than 18 decimals.

Code Snippet

Constants.precision is `1e18`:

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Libs/Constants.sol#L24

MIN_COL_RATIO is expressed in 1e18:

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L57

TauDripFeed uses `1e18` once again in calculation with collateral amount:

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/TauDripFeed.sol#L91

The calculation for maxRepay is also impacted:

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L251-L253

The calculation for expected liquidation collateral is also impacted:

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L367

Tool used

Manual Review

Recommendation

Account for the collateral decimals in the calculation instead of using `Constants.PRECISION`.

Discussion

IAm0x52

Escalate for 10 USDC

This should be medium rather than high, since affected tokens would simply be incompatible because MIN_COL_RATIO is expressed to 18 dp.

sherlock-admin

> Escalate for 10 USDC
>
> This should be medium rather than high, since affected tokens would simply be incompatible because MIN_COL_RATIO is expressed to 18 dp.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

iHarishKumar

https://github.com/protokol/taurus-contracts/pull/124

spyrosonic10

Escalate for 10 USDC

From the constest scope `ERC20: any non-rebasing. In particular, fee + staked GLP will be the first collateral token (managed through GMX's ERC20-compliant wrapper) and Arbitrum Weth will be the main yield token.`

Present state of contract is designed and to use with 18 decimal gmx token. Future release should be out of scope so it is invalid issue. In order to launch other vault types protocol will be the first to know about decimal things. Given the current state of protocol and codebase this issue doesn't pose any risk to user funds. Not qualified for high.

sherlock-admin

> Escalate for 10 USDC
>
> From the constest scope `ERC20: any non-rebasing. In particular, fee + staked GLP will be the first collateral token (managed through GMX's ERC20-compliant wrapper) and Arbitrum Weth will be the main yield token.`
>
> Present state of contract is designed and to use with 18 decimal gmx token. Future release should be out of scope so it is invalid issue. In order to launch other vault types protocol will be the first to know about decimal things. Given the current state of protocol and codebase this issue doesn't pose any risk to user funds. Not qualified for high.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

hrishibhat

Escalation rejected

Given that the assumption for 18 decimals impacts calculations across the codebase and also affects the borrow calculations, considering this issue a valid high.

sherlock-admin

> Escalation rejected
>
> Given that the assumption for 18 decimals impacts calculations across the codebase and also affects the borrow calculations, considering this issue a valid high.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.

MLON33

> https://github.com/protokol/taurus-contracts/pull/124

Fixed here

IAm0x52

Fix allows tokens without 18 dp to be used but liquidations can now cause dust to accumulate in contract

jacksanford1

Protocol team (Meriadoc) comment from Discord:

> Yeah I will also look into it and double check it doesn't lead to any problems further down the line, but as is that seems fine.

The dust amount will be considered "acknowledged" by the protocol team.

Issue H-2: Missing input validation for _rewardProportion parameter allows keeper to escalate his privileges and pay back all loans

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/11

Found by

cducrest-brainbot, roguereddwarf

Summary

According to the Contest page and discussion with the sponsor, the role of a `keeper` is to perform liquidations and to swap yield token for `TAU` using the `SwapHandler.swapForTau` function: https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/SwapHandler.sol#L45-L52

They are also able to choose how much yield token to swap and what the proportion of the resulting TAU is that is distributed to users vs. not distributed in order to erase bad debt.

So a `keeper` is not trusted to perform any actions that go beyond swapping yield / performing liquidations.

However there is a missing input validation for the `_rewardProportion` parameter in the `SwapHandler.swapForTau` function. This allows a keeper to "erase" all debt of users. So users can withdraw their collateral without paying any of the debt.

Vulnerability Detail

By looking at the code we can see that `_rewardProportion` is used to determine the amount of `TAU` that `_withholdTau` is called with: Link

```
_withholdTau((tauReturned * _rewardProportion) / Constants.PERCENT_PRECISION);
```

Any value of `_rewardProportion` greater than `1e18` means that more `TAU` will be distributed to users than has been burnt (aka erasing debt).

It is easy to see how the `keeper` can chose the number so big that `_withhholdTau` is called with a value close to `type(uint256).max` which will certainly be enough to erase all debt.

Impact

A `keeper` can escalate his privileges and erase all debt. This means that `TAU` will not be backed by any collateral anymore and will be worthless.

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/SwapHandler.sol#L45-L101

Tool used

Manual Review

Recommendation

I discussed this issue with the sponsor and it is intended that the `keeper` role can freely chose the value of the `_rewardProportion` parameter within the `[0,1e18]` range, i.e. 0%-100%.

Therefore the fix is to simply check that `_rewardProportion` is not bigger than `1e18`:

```
diff --git a/taurus-contracts/contracts/Vault/SwapHandler.sol
 ↪  b/taurus-contracts/contracts/Vault/SwapHandler.sol
index c04e3a4..ab5064b 100644
--- a/taurus-contracts/contracts/Vault/SwapHandler.sol
+++ b/taurus-contracts/contracts/Vault/SwapHandler.sol
@@ -59,6 +59,10 @@ abstract contract SwapHandler is FeeMapping, TauDripFeed {
             revert zeroAmount();
         }

+        if (_rewardProportion > Constants.PERCENT_PRECISION) [
+            revert invalidRewardProportion();
+        ]
+
         // Get and validate swap adapter address
         address swapAdapterAddress =
 ↪  SwapAdapterRegistry(controller).swapAdapters(_swapAdapterHash);
         if (swapAdapterAddress == address(0)) {
```

Discussion

Sierraescape

https://github.com/protokol/taurus-contracts/pull/121

MLON33

> https://github.com/protokol/taurus-contracts/pull/121

Fixed here

IAm0x52

Fix looks good. swapForTau will now revert if keeper specifies a _rewardProportion that is too large

Issue M-1: `swap()` will be reverted if `path` has more tokens.

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/160

Found by

GimelSec, shaka

Summary

`swap()` will be reverted if `path` has more tokens, the keepers will not be able to successfully call `swapForTau()`.

Vulnerability Detail

In `test/SwapAdapters/00_UniswapSwapAdapter.ts`:

```
// Get generic swap parameters
const basicSwapParams = buildUniswapSwapAdapterData(
  ["0xyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy",
↪   "0xzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz"],
  [3000],
  testDepositAmount,
  expectedReturnAmount,
  0,
).swapData;
```

We will get:

```
00000000000000000000000000000000000000000000000000000000024f49cbca
00000000000000000000000000000000000000000000000056bc75e2d63100000
00000000000000000000000000000000000000000000000055de6a779bbac0000
0000000000000000000000000000000000000000000000000000000000000080
000000000000000000000000000000000000000000000000000000000000002b
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy000bb8zzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzzzzzz000000000000000000000000000000000000000000
```

Then the `swapOutputToken` is `_swapData[length - 41:length - 21]`.

But if we have more tokens in path:

```
const basicSwapParams = buildUniswapSwapAdapterData(
  ["0xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
↪   "0xyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy",
↪   "0xzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz"],
  [3000, 3000],
  testDepositAmount,
  expectedReturnAmount,
  0,
).swapData;
```

```
000000000000000000000000000000000000000000000000000000024f49cbca
00000000000000000000000000000000000000000000000056bc75e2d63100000
00000000000000000000000000000000000000000000000055de6a779bbac0000
0000000000000000000000000000000000000000000000000000000000000080
0000000000000000000000000000000000000000000000000000000000000042
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx000bb8yyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyy000bb8zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
zzzz000000000000000000000000000000000000000000000000000000000000
```

swapOutputToken is _swapData[length - 50:length - 30], the swap() function will be reverted.

Impact

The keepers will not be able to successfully call SwapHandler.swapForTau(). Someone will get a reverted transaction if they misuse UniswapSwapAdapter.

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/SwapAdapters/UniswapSwapAdapter.sol#L30

Tool used

Manual Review

Recommendation

Limit the swap pools, or check if the balance of _outputToken should exceed _amountOutMinimum.

Discussion

Sierraescape

https://github.com/protokol/taurus-contracts/pull/82

MLON33

> https://github.com/protokol/taurus-contracts/pull/82

Fixed here

IAm0x52

Fix looks good. Byte position of outputToken is now calculated dynamically based on path length rather than a hardcoded 21 bytes which only worked for single hops

Issue M-2: Mint limit is not reduced when the Vault is burning TAU

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/149

Found by

8olidity, Chinmay, GimelSec, HonorLt, LethL, Ruhum, SunSec, bytes032, cducrest-brainbot, chaduke, duc, mstpr-brainbot, nobody2018, shaka, tvdung94, y1cunhui

Summary

Upon burning TAU, it incorrectly updates the `currentMinted` when Vault is acting on behalf of users.

Vulnerability Detail

When the burn of `TAU` is performed, it calls `_decreaseCurrentMinted` to reduce the limit of tokens minted by the Vault:

```
function _decreaseCurrentMinted(address account, uint256 amount) internal
↪   virtual {
    // If the burner is a vault, subtract burnt TAU from its currentMinted.
    // This has a few highly unimportant edge cases which can generally be
↪   rectified by increasing the relevant vault's mintLimit.
    uint256 accountMinted = currentMinted[account];
    if (accountMinted >= amount) {
        currentMinted[msg.sender] = accountMinted - amount;
    }
}
```

The issue is that it subtracts `accountMinted` (which is `currentMinted[account]`) from `currentMinted[msg.sender]`. When the vault is burning tokens on behalf of the user, the `account` != `msg.sender` meaning the `currentMinted[account]` is 0, and thus the `currentMinted` of Vault will be reduced by 0 making it pretty useless.

Another issue is that users can transfer their `TAU` between accounts, and then `amount > accountMinted` will not be triggered.

Impact

`currentMinted` is incorrectly decreased upon burning so vaults do not get more space to mint new tokens.

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/TAU.sol#L76-L83

Tool used

Manual Review

Recommendation

A simple solution would be to:

```
uint256 accountMinted = currentMinted[msg.sender];
```

But I suggest revisiting and rethinking this function altogether.

Discussion

Sierraescape

https://github.com/protokol/taurus-contracts/pull/85

MLON33

> https://github.com/protokol/taurus-contracts/pull/85

Fixed here

IAm0x52

Fix looks good. _decreaseCurrentMinted now always uses msg.sender as account

Issue M-3: Account can not be liquidated when price fall by 99%.

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/61

Found by

roguereddwarf, spyrosonic10

Summary

Liquidation fails when price fall by 99%.

Vulnerability Detail

`_calcLiquidation()` method has logic related to liquidations. This method calculate total liquidation discount, collateral to liquidate and liquidation surcharge. All these calculations looks okay in normal scenarios but there is an edge case when liquidation fails if price crashes by 99% or more. In such scenario `collateralToLiquidateWithoutDiscount` will be very large and calculated liquidation surcharge becomes greater than `collateralToLiquidate`

```
uint256 collateralToLiquidateWithoutDiscount = (_debtToLiquidate * (10 **
↪   decimals)) / price;
collateralToLiquidate = (collateralToLiquidateWithoutDiscount *
↪   totalLiquidationDiscount) / Constants.PRECISION;
if (collateralToLiquidate > _accountCollateral) {
            collateralToLiquidate = _accountCollateral;
}
uint256 liquidationSurcharge = (collateralToLiquidateWithoutDiscount *
↪   LIQUIDATION_SURCHARGE) / Constants.PRECISION
```

Contract revert from below line hence liquidation will fail in this scenario.

```
uint256 collateralToLiquidator = collateralToLiquidate - liquidationSurcharge;
```

Impact

Liquidation fails when price crash by 99% or more. Expected behaviour is that liquidation should be successful in all scenarios.

Code Snippet

Block of code that has bug. https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L396-L422

Below is POC that prove failed liquidation.

```
it("should revert liquidation if an account is unhealthy and price crashed 99%", async () => {
        // Assume price is crashed 99%
        await glpOracle.updatePrice(PRECISION.mul(1).div(100));
        // check if the account is underwater
        const health = await gmxVault.getAccountHealth(user.address);
        expect(health).eq(false);

        // Check the liquidation amount
        const liqAmt = await gmxVault.getMaxLiquidation(user.address);

        // Mint some TAU to the liquidator and approve vault to spend it
        await mintHelper(liqAmt, liquidator.address);
        await tau.connect(liquidator).approve(gmxVault.address, liqAmt);
        const totalTauSupply = await tau.totalSupply();

        // liquidation will fail
        const tx = gmxVault.connect(liquidator).liquidate(user.address, liqAmt, 0);
        // reverted with panic code 0x11 (Arithmetic operation underflowed or overflowed outside of an unchecked block)
        await expect(tx).revertedWithPanic(0x11);
      });
```

PS: This test goes in 00_GmxYieldAdapter.ts and inside describe("Liquidate", async () => { block defined at line 269

Tool used

Manual Review

Recommendation

Presently liquidation surcharge is calculated on `collateralToLiquidateWithoutDiscount`. Project team may want to reconsider this logic and calculate surcharge on `collateralToLiquidate` instead of `collateralToLiquidateWithoutDiscount`. This will be business decision but easy fix

Another option is you may want to calculate surcharge on `Math.min(collateralToLiquidate, collateralToLiquidateWithoutDiscount)`.

```
uint256 collateralToTakeSurchargeOn = Math.min(collateralToLiquidate,
 ↪  collateralToLiquidateWithoutDiscount);
uint256 liquidationSurcharge = (collateralToTakeSurchargeOn *
 ↪  LIQUIDATION_SURCHARGE) / Constants.PRECISION;
return (collateralToLiquidate, liquidationSurcharge);
```

Discussion

Sierraescape

https://github.com/protokol/taurus-contracts/pull/122

hrishibhat

Since this is an edge case for the given price fall resulting in reverting liquidations, Considering this as a valid medium

IAm0x52

Escalate for 10 USDC

This is the same root cause as #89 that the liquidation surcharge is calculated based on the uncapped amount. This is another symptom of that same underlying problem, so it should be a dupe of #89

sherlock-admin

> Escalate for 10 USDC
>
> This is the same root cause as #89 that the liquidation surcharge is calculated based on the uncapped amount. This is another symptom of that same underlying problem, so it should be a dupe of #89

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

spyrosonic10

Escalate for 10 USDC

I do not agree with escalation raised above. This issue is about failure of liquidation when price fall by x%. This finding is an edge case where it does impact all underwater accounts so it is fair to say that it impact whole protocol. Root cause and impact both are different in this issue compare to #89 so this is definitely not a duplicate of #89.

sherlock-admin

> Escalate for 10 USDC
>
> I do not agree with escalation raised above. This issue is about failure of liquidation when price fall by x%. This finding is an edge case where it does impact all underwater accounts so it is fair to say that it impact whole protocol. Root cause and impact both are different in this issue compare to #89 so this is definitely not a duplicate of #89.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

hrishibhat

Escalation accepted

Accepting the first escalation. After further internal discussion, both the outcomes originate out of the same root cause of using `collateralToLiquidateWithoutDiscount` to calculate `liquidationSurcharge`. While one mentions increase in the fee the other instance increases to cause underflow. Considering #89 a duplicate of this issue.

sherlock-admin

> Escalation accepted
>
> Accepting the first escalation. After further internal discussion, both the outcomes originate out of the same root cause of using `collateralToLiquidateWithoutDiscount` to calculate `liquidationSurcharge`. While one mentions increase in the fee the other instance increases to cause underflow. Considering #89 a duplicate of this issue.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

MLON33

> https://github.com/protokol/taurus-contracts/pull/122

Fixed here

IAm0x52

Fix looks good. Surcharge is now taken on actual amount liquidated rather than undercounted amount

Issue M-4: A malicious admin can steal all users collateral

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/43

Found by

J4de, KingNFT

Summary

According to Taurus contest details, all roles, including the admin `Multisig`, should not be able to drain users collateral.

```
2. Multisig. Trusted with essentially everything but user collateral.
```

https://app.sherlock.xyz/audits/contests/45 But the current implementation allows admin to update price feed without any restriction, such as `timelock`. This leads to an attack vector that a malicious admin can steal all users collateral.

Vulnerability Detail

As shown of `updateWrapper()` function of `PriceOracleManager.sol`, the admin (`onlyOwner`) can update any price oracle `_wrapperAddress` for any `_underlying` collateral without any restrictions (such as `timelock`).

```
File: taurus-contracts\contracts\Oracle\PriceOracleManager.sol
    function updateWrapper(address _underlying, address _wrapperAddress) external override onlyOwner {
        if (!_wrapperAddress.isContract()) revert notContract();
        if (wrapperAddressMap[_underlying] == address(0)) revert wrapperNotRegistered(_wrapperAddress);

        wrapperAddressMap[_underlying] = _wrapperAddress;

        emit WrapperUpdated(_underlying, _wrapperAddress);
    }
```

Hence, admin can set a malicious price oracle like

```
contract AttackOracleWrapper is IOracleWrapper, Ownable {
    address public attacker;
    IGLPManager public glpManager;

    constructor(address _attacker, address glp) {
        attacker = _attacker;
        glpManager = IGLPManager(glp);
    }

    function getExternalPrice(
        address _underlying,
        bytes calldata _flags
    ) external view returns (uint256 price, uint8 decimals, bool success) {
        if (tx.origin == attacker) {
            return (1, 18, true); // @audit a really low price resulting in the liquidation of all positions
        } else {
            uint256 price = glpManager.getPrice();
            return (price, 18, true);
        }
    }
}
```

Then call `liquidate()` to drain out users collateral with negligible $TAU cost.

```
File: taurus-contracts\contracts\Vault\BaseVault.sol
    function liquidate(
        address _account,
        uint256 _debtAmount,
        uint256 _minExchangeRate
    ) external onlyLiquidator whenNotPaused updateReward(_account) returns (bool) {
        if (_debtAmount == 0) revert wrongLiquidationAmount();

        UserDetails memory accDetails = userDetails[_account];

        // Since Taurus accounts' debt continuously decreases, liquidators may pass in an arbitrarily large number in order to
        // request to liquidate the entire account.
        if (_debtAmount > accDetails.debt) {
            _debtAmount = accDetails.debt;
        }

        // Get total fee charged to the user for this liquidation. Collateral equal to (liquidated taurus debt value * feeMultiplier) will be deducted from the user's account.
        // This call reverts if the account is healthy or if the liquidation amount is too large.
        (uint256 collateralToLiquidate, uint256 liquidationSurcharge) = _calcLiquidation(
            accDetails.collateral,
            accDetails.debt,
            _debtAmount
        );

        // Check that collateral received is sufficient for liquidator
        uint256 collateralToLiquidator = collateralToLiquidate - liquidationSurcharge;
        if (collateralToLiquidator < (_debtAmount * _minExchangeRate) / Constants.PRECISION) {
            revert insufficientCollateralLiquidated(_debtAmount, collateralToLiquidator);
        }

        // Update user info
        userDetails[_account].collateral = accDetails.collateral - collateralToLiquidate;
        userDetails[_account].debt = accDetails.debt - _debtAmount;

        // Burn liquidator's Tau
        TAU(tau).burnFrom(msg.sender, _debtAmount);

        // Transfer part of _debtAmount to liquidator and Taurus as fees for liquidation
        IERC20(collateralToken).safeTransfer(msg.sender, collateralToLiquidator);
        IERC20(collateralToken).safeTransfer(
            Controller(controller).addressMapper(Constants.FEE_SPLITTER),
            liquidationSurcharge
        );

        emit AccountLiquidated(msg.sender, _account, collateralToLiquidate, liquidationSurcharge);

        return true;
    }
```

Impact

A malicious admin can steal all users collateral

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L342

Tool used

Manual Review

Recommendation

update of price oracle should be restricted with a `timelock`.

Discussion

iHarishKumar

https://github.com/protokol/taurus-contracts/pull/128

spyrosonic10

Escalate for 10 USDC

PriceOracleManger is Ownable contract so yes it has `owner` param and not `governor` param. So here owner can be governor, timelock and multisig. Also when it come to calling `updateWrapper` multisig can be trusted as it can be trusted to not set deposit fee to max and loot all users. So with that being said this is info/low issue and does not qualify for medium. It may be possible that is entirely out of scope as it is related to admin controlled param.

sherlock-admin

> Escalate for 10 USDC
>
> PriceOracleManger is Ownable contract so yes it has `owner` param and not `governor` param. So here owner can be governor, timelock and multisig. Also when it come to calling `updateWrapper` multisig can be trusted as it can be trusted to not set deposit fee to max and loot all users. So with that being said this is info/low issue and does not qualify for medium. It may be possible that is entirely out of scope as it is related to admin controlled param.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

hrishibhat

Escalation rejected

Given that the protocol clearly mentions that admin should be restricted whenever possible from affecting the user collateral adding the restriction makes sense. Considering this issue a valid medium.

sherlock-admin

> Escalation rejected
>
> Given that the protocol clearly mentions that admin should be restricted whenever possible from affecting the user collateral adding the restriction makes sense. Considering this issue a valid medium.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.

MLON33

> https://github.com/protokol/taurus-contracts/pull/128

Fixed here

IAm0x52

Fix looks good. Timelocker will now be set as owner of oracle manager adding a timelock to any oracle change

Issue M-5: SwapHandler.sol: Check that collateral token cannot be swapped is insufficient for tokens with multiple addresses

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/31

Found by

rogueredwarf

Summary

According to the contest page `any non-rebasing` ERC20 token is supposed to be supported.

The `SwapHandler.swapForTau` function checks that the `collateralToken` cannot be sent to the `SwapAdapter` for trading:

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/SwapHandler.sol#L54-L56

Vulnerability Detail

There exist however ERC20 tokens that have more than one address. In case of such a token, the above check is not sufficient. The token could be swapped anyway by using a different address.

Impact

The check that collateral cannot be swapped can be bypassed for tokens with multiple addresses.

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/SwapHandler.sol#L45-L101

Tool used

Manual Review

Recommendation

Compare the balance of the collateral before and after sending tokens to the SwapAdapter and make sure it hasn't changed. Or implement a whitelist for tokens that can be swapped.

Discussion

Sierraescape

Tokens with multiple addresses are pretty rare, so we're just going to note that the vault doesn't allow such tokens as collateral, and create wrappers for them if necessary.

https://github.com/protokol/taurus-contracts/pull/120

spyrosonic10

Escalate for 10 USDC

Token with different addresses is very very rare. Almost every protocols in Defi operating on assumption of token with single address. This issue does not qualify as High/Medium.

sherlock-admin

> Escalate for 10 USDC
>
> Token with different addresses is very very rare. Almost every protocols in Defi operating on assumption of token with single address. This issue does not qualify as High/Medium.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

roguereddwarf

Escalate for 10 USDC

Disagree with previous escalation. While these tokens are rare they do exist and as pointed out in my report `any non-rebasing` ERC20 is supposed to be supported which clearly includes tokens with multiple addresses. So I think this is a valid medium.

sherlock-admin

> Escalate for 10 USDC
>
> Disagree with previous escalation. While these tokens are rare they do exist and as pointed out in my report `any non-rebasing` ERC20 is supposed to be supported which clearly includes tokens with multiple addresses. So I think this is a valid medium.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

hrishibhat

Escalation accepted

Considering this a valid medium. As pointed out in the second escalation, even though these tokens are rare the issue can still be considered valid medium.

Note: Going forward, Sherlock team will add additional clarity on such rare token cases in the README.

sherlock-admin

> Escalation accepted
>
> Considering this a valid medium. As pointed out in the second escalation, even though these tokens are rare the issue can still be considered valid medium.
>
> Note: Going forward, Sherlock team will add additional clarity on such rare token cases in the README.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

MLON33

> Tokens with multiple addresses are pretty rare, so we're just going to note that the vault doesn't allow such tokens as collateral, and create wrappers for them if necessary.
>
> https://github.com/protokol/taurus-contracts/pull/120

Fixed here

IAm0x52

No direct fix has been implemented but a note has been added explicitly stating that multi-address tokens are not supported

jacksanford1

Classifying this issue as "Acknowledged" since no direct fix was made.

Issue M-6: User can prevent liquidations by frontrunning the tx and slightly increasing their collateral

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/12

Found by

Ruhum, cryptostellar5

Summary

User can prevent liquidations by frontrunning the tx and decreasing their debt so that the liquidation transaction reverts.

Vulnerability Detail

In the liquidation transaction, the caller has to specify the amount of debt they want to liquidate, `_debtAmount`. The maximum value for that parameter is the total amount of debt the user holds:

In `_calcLiquidation()`, the contract determines how much collateral to liquidate when `_debtAmount` is paid by the caller. In that function, there's a check that reverts if the caller tries to liquidate more than they are allowed to depending on the position's health.

The goal is to get that if-clause to evaluate to `true` so that the transaction reverts. To modify your position's health you have two possibilities: either you increase your collateral or decrease your debt. So instead of preventing the liquidation by pushing your position to a healthy state, you only modify it slightly so that the caller's liquidation transaction reverts.

Given that Alice has:

- 100 TAU debt
- 100 Collateral (price = $1 so that collateralization rate is 1) Her position can be liquidated. The max value is:

$(1.3e18 * 100e18 - (100e18 * 1e18 * 1e18)/1e18)/1.3e18 = 23.07e18$ (leave out liquidation discount for easier math)

The liquidator will probably use the maximum amount they can liquidate and call `liquidate()` with `23.07e18`. Alice frontruns the liquidator's transaction and increases the collateral by `1`. That will change the max liquidation amount to: $(1.3e18 * 100e18 - 101e18 * 1e18)/1.3e18 = 22.3e18$.

That will cause `_calcLiquidation()` to revert because `23.07e18 > 22.3e18`.

The actual amount of collateral to add or debt to decrease depends on the liquidation transaction. But, generally, you would expect the liquidator to liquidate as much as possible. Thus, you only have to slightly move the position to cause their transaction to revert

Impact

User can prevent liquidations by slightly modifying their position without putting it at a healthy state.

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L342-L363

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L396-L416

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L240

Tool used

Manual Review

Recommendation

In `_calcLiquidation()` the function shouldn't revert if `_debtToLiqudiate > _getMaxLiquidation()`. Instead, just continue with the value `_getMaxLiquidation()` returns.

Discussion

Sierraescape

Great write-up and recommendation.

Sierraescape

https://github.com/protokol/taurus-contracts/pull/115

IAm0x52

Escalate for 10 USDC

Should be medium since it can only prevent the liquidation temporarily. Similar issues have always been given medium severity such as Cooler:

https://github.com/sherlock-audit/2023-01-cooler-judging/issues/218

sherlock-admin

> Escalate for 10 USDC
>
> Should be medium since it can only prevent the liquidation temporarily. Similar issues have always been given medium severity such as Cooler:
>
> https://github.com/sherlock-audit/2023-01-cooler-judging/issues/218

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

spyrosonic10

Escalate for 10 USDC

Double down the escalation raise point the it only prevent the liquidation temporarily and hence it should be medium.

sherlock-admin

> Escalate for 10 USDC
>
> Double down the escalation raise point the it only prevent the liquidation temporarily and hence it should be medium.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

hrishibhat

Escalation accepted

Considering the impact of the attack is just preventing liquidations temporarily, considering this a valid medium

sherlock-admin

> Escalation accepted
>
> Considering the impact of the attack is just preventing liquidations temporarily, considering this a valid medium

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

MLON33

https://github.com/protokol/taurus-contracts/pull/115

Fixed here

IAm0x52

Fix looks good. _calcLiquidation now returns a lower debt amount if user is liquidating more than max collateral