SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Taurus
Prepared by:	Sherlock
Lead Security Expert:	0x52
Dates Audited:	March 6 - March 13, 2023
Prepared on:	April 1, 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

taurus-contracts @ 3759a646f5738890198eb7ae3964e4ecbe952d17

- taurus-contracts/contracts/Controller/Controllable.sol
- taurus-contracts/contracts/Controller/ControllableUpgradeable.sol
- taurus-contracts/contracts/Controller/Controller.sol
- taurus-contracts/contracts/Controller/SwapAdapterRegistry.sol
- taurus-contracts/contracts/Libs/Constants.sol
- taurus-contracts/contracts/Libs/ParseBytes.sol
- taurus-contracts/contracts/Libs/TauMath.sol
- taurus-contracts/contracts/LiquidationBot/LiquidationBot.sol
- taurus-contracts/contracts/Oracle/CustomPriceOracle/CustomPriceOracle.sol
- taurus-contracts/contracts/Oracle/CustomPriceOracle/GLPPriceOracle.sol
- taurus-contracts/contracts/Oracle/OracleInterface/IGLPManager.sol
- taurus-contracts/contracts/Oracle/OracleInterface/IOracleWrapper.sol
- taurus-contracts/contracts/Oracle/OracleInterface/IPriceOracle.sol
- taurus-contracts/contracts/Oracle/OracleInterface/IPriceOracleManager.sol
- taurus-contracts/contracts/Oracle/PriceOracleManager.sol
- taurus-contracts/contracts/Oracle/Wrapper/CustomOracleWrapper.sol
- taurus-contracts/contracts/SwapAdapters/BaseSwapAdapter.sol
- taurus-contracts/contracts/SwapAdapters/ISwapRouter02.sol
- taurus-contracts/contracts/SwapAdapters/UniswapSwapAdapter.sol
- taurus-contracts/contracts/TAU.sol
- taurus-contracts/contracts/TGT.sol
- taurus-contracts/contracts/Tokenomics/FeeSplitter.sol
- taurus-contracts/contracts/Vault/BaseVault.sol
- taurus-contracts/contracts/Vault/FeeMapping.sol

- taurus-contracts/contracts/Vault/SwapHandler.sol
- taurus-contracts/contracts/Vault/TauDripFeed.sol
- taurus-contracts/contracts/Vault/YieldAdapters/GMX/GmxYieldAdapter.sol
- taurus-contracts/contracts/Vault/YieldAdapters/GMX/IRewardRouter.sol

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
12	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x52	libratus	Bahurum
roguereddwarf	Diana	RaymondFam
cducrest-brainbot	Chinmay	peanuts
Ruhum	Ityu	nobody2018
cryptostellar5	bytes032	8olidity
spyrosonic10	duc	tvdung94
GimelSec	mstpr-brainbot	y1cunhui
KingNFT	Bauer	HonorLt
yixxas	jonatascm	SunSec
shaka	imare	LethL
J4de	ck	chaduke

Issue H-1: Protocol assumes 18 decimals collateral

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

Found by

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

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`.

Issue H-2: 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

Issue H-3: 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 `_withholdTau` 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

Issue M-1: GmxYieldAdapter#collectYield continues to function even on a paused vault

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/182

Found by

0x52

Summary

In the readme it states: "Users can exit paused vaults, but otherwise no significant action should be possible on them." However, it is still possible to collect yield from GMX when the vault is paused. This is because GmxYieldAdapter#collectYield lacks the whenNotPaused modifier.

Vulnerability Detail

See summary

Impact

Yield can still be collected even when the vault is paused, which is problematic depending on why the vault is paused

Code Snippet

GmxYieldAdapter.sol#L32-L35

Tool used

Manual Review

Recommendation

Add the whenNotPaused modifer to GmxYieldAdapter#collectYield

Discussion

Sierraescape

https://github.com/protokol/taurus-contracts/pull/83

Issue M-2: `swap()` will be reverted if `path` has more tokens.

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

Found by

shaka, GimelSec

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:

```
000000000000000000000000000000000000000000000000000000024f49cbca
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

Issue M-3: 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

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

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

Issue M-4: Funds can be stolen from `UniswapSwapAdapter` if swap was satisfied only partially

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/133

Found by

libratus, roguereddwarf

Summary

There can be scenarios where tokens transferred to `UniswapSwapAdapter` are only partially swapped for TAU. In that case, a certain amount remains on the contract and can be withdrawn by anyone

Vulnerability Detail

In order to swap yield rewards for TAU `SwapHandler.swapForTau` function transfers yield tokens to `UniswapSwapAdapter` and calls `swap`.

```
IERC20(_yieldTokenAddress).safeTransfer(swapAdapterAddress, swapAmount);

// Call swap function, which will transfer resulting tau back to this contract
↪  and return the amount transferred.
// Note that this contract does not check that the swap adapter has transferred
↪  the correct amount of tau. This check
// is handled by the swap adapter, and for this reason any registered swap
↪  adapter must be a completely trusted contract.
uint256 tauReturned = BaseSwapAdapter(swapAdapterAddress).swap(tau, _swapParams);
```

The amount of tokens to swap is passed in `_yieldTokenAmount` parameters, while swap parameters are in `_swapParams`.

```
function swapForTau(
    address _yieldTokenAddress,
    uint256 _yieldTokenAmount,
    uint256 _minTauReturned,
    bytes32 _swapAdapterHash,
    uint256 _rewardProportion,
    bytes calldata _swapParams
) external onlyKeeper whenNotPaused {
```

`UniswapSwapAdapter` is supposed to perform a swap using *all* of the yield tokens sent to it and return "out" tokens back to `SwapHandler`. There can be scenarios where not

SHERLOCK

all yield tokens are swapped, in which case they will remain on `UniswapSwapAdapter` and will be available for withdrawal by anyone since `swap` has no access control.

First scenario is if there was an inconsistency between `_yieldTokenAmount` and `_swapParams`. These parameters are not validated against each other. It is possible to pass `swapParams` with `_amountIn` much smaller than `_yieldTokenAmount`. It is an input validation issue, but one that leads to instant loss of funds and therefore can be considered "Medium".

Second scenario is in an unlikely case that liquidity pool is depleted and there are not enough TAU tokens to perform the swap fully. In this case, Uniswap will perform a partial swap and the unswapped yield tokens will remain on the contract.

Impact

Tokens can be stolen from `UniswapSwapAdapter`

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/SwapHandler.sol#L75-L80

Tool used

Manual Review

Recommendation

- validate `_yieldTokenAmount` and `_swapParams` against each other
- return tokens back to `SwapHandler` in case of a partial Uniswap swap
- consider restricting access for `UniswapSwapAdapter.swap`.

Discussion

Sierraescape

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

Issue M-5: baseVault#emergencyClosePosition permanently breaks award accounting

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/117

Found by

yixxas, 0x52, GimelSec

Summary

When using emergency close it should store the unclaimed rewards to be distributed later since the user can't claim them later

Vulnerability Detail

BaseVault.sol#L227-L229

```
function emergencyClosePosition() external {
    _modifyPosition(msg.sender, userDetails[msg.sender].collateral,
    ↪  userDetails[msg.sender].debt, false, false);
}
```

`emergencyClosePosition` allows a user to completely close their position without updating their debt. This allows users to force close their position even when the contract is paused. Because it bypasses `updateReward`, all unclaimed TAU will be silently lost and becomes unrecoverable.

BaseVault.sol#L78-L118

```
modifier updateReward(address _account) {
    // Disburse available yield from the drip feed
    _disburseTau();

    // If user has collateral, pay down their debt and recycle surplus rewards
    ↪  back into the tauDripFeed.
    uint256 _userCollateral = userDetails[_account].collateral;
    if (_userCollateral > 0) {
        ...
    } else {
        // If this is a new user, add them to the userAddresses array to keep
        ↪  track of them.
        if (userDetails[_account].startTimestamp == 0) {
            userAddresses.push(_account);
            userDetails[_account].startTimestamp = block.timestamp;
        }
    }

    // Update user lastUpdatedRewardPerCollateral
    userDetails[_account].lastUpdatedRewardPerCollateral =
     ↪  cumulativeTauRewardPerCollateral;
    _;
}
```

Since `emergencyClosePosition` completely closes out the entire position the next time a user interacts with the contract it will bypass all the reward logic and update their `lastUpdatedRewardPerCollateral`. This means that all accrued rewards before calling `emergencyClosePosition` can never be claimed.

Example:

A user's `rewardPerCollateral` is 1e18 and the `cumulativeRewardPerCollateral` is 1.1e18 with a collateral of 100e18. This would mean that the user is owed 10e18 in rewards. They call `emergencyClosePosition` and now that 10e18 is lost. Since this value is never used to offset any debt it is essentially lost.

Impact

TAU will be silently lost and becomes unrecoverable.

Code Snippet

BaseVault.sol#L78-L118

Tool used

Manual Review

Recommendation

Lost rewards should be tracked distributed to the rest of the vault.

Discussion

Sierraescape

> TAU will be silently lost and becomes unrecoverable.

Since it is burned, it would be more accurate to say that tau is returned to the protocol rather than used to pay off user debt. This is unfortunate for the users, but the alternative is to risk tau hyperinflation if the vault is paused due to an issue with the reward accounting system.

We will make it clear to users that if they withdraw while the vault is paused, they will not earn any unclaimed rewards. If this is a large issue, we'd prefer to mint tau directly to affected users (via a governance action once whatever circumstances paused the vault have been resolved) rather than open the vault up to this sort of vulnerability.

So this is intended behavior.

Evert0x

Still considering medium as it seems like it can be used for a grieving attack.

Issue M-6: BaseVault: liquidationSurcharge amount is too high if collateralToLiquidate gets capped

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/89

Found by

roguereddwarf

Summary

According to the comment explaining the `LIQUIDATION_SURCHARGE` percentage, it should be a percentage of the liquidated amount: Link

```
// 2% of the liquidated amount is sent to the FeeSplitter to fund protocol
↪  stability
// and disincentivize self-liquidation
```

So if say $100 of collateral is liquidated, the surcharge should be $2. I will show how this is not true in some cases. I.e. the surcharge may be higher (even a lot higher).

I had a discussion about this with the sponsor and it was assessed that this is unintended behavior. The liquidation surcharge percentage breaks in the case that I will show.

Vulnerability Detail

Let's have a look at how the liquidation surcharge is calculated: Link

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


// Revert if requested liquidation amount is greater than allowed
if (
    _debtToLiquidate >
    _getMaxLiquidation(_accountCollateral, _accountDebt, price, decimals,
↪  totalLiquidationDiscount)
) revert wrongLiquidationAmount();
```

```
return (
    collateralToLiquidate,
    (collateralToLiquidateWithoutDiscount * LIQUIDATION_SURCHARGE) /
↪   Constants.PRECISION
);
```

The `collateralToLiquidate` is calculated by multiplying `collateralToLiquidateWithoutDiscount` by `totalLiquidationDiscount`. And then it is capped at `_accountCollateral`. The cap is necessary because it is not possible to remove more collateral from the loan than there is left.

As part of the return statement, the `liquidationSurcharge` return value is calculated as `(collateralToLiquidateWithoutDiscount * LIQUIDATION_SURCHARGE) / Constants.PRECISION`.

It becomes clear that this can result in a higher liquidation surcharge than expected: Let `collateralToLiquidateWithoutDiscount=$100`, `totalLiquidationDiscount=120%` (20% percent discount on the collateral). Then `collateralToLiquidate=$100*120%=$120` and `liquidationSurcharge=$100*2%=$2`.

If however the `collateralToLiquidate` gets capped at say
50, *then‘liquidationSurcharge‘isstillcalculatedinthesameway.Butthesurchargepercentagewouldactuallyt*
instead of the expected 2%.

Impact

The `liquidationSurcharge` that is charged to the liquidator by subtracting it from the collateral he receives may not be the 2% that it is supposed to. This means that the liquidator needs to pay a higher liquidation fee in some cases and ends up with less profit.

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L396-L422

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L432-L445

Tool used

Manual Review

Recommendation

It is not obvious how this issue can be fixed because the surcharge percentage is not well defined for when `collateralToLiquidate > _accountCollateral` and the `collateralToLiquidate` is capped.

After all if this is the case then what should `collateralToLiquidateWithoutDiscount` be? Should the amount by which `collateralToLiquidate` is reduced due to the cap be taken from the discount or from `collateralToLiquidate` AND the discount.

In other words it is not clear what value to use to calculate the liquidation surcharge.

This has been discussed with the sponsor and they should look into different ways to modify the calculation by first firmly establishing how the liquidation surcharge is intended to behave.

Discussion

Sierraescape

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

Issue M-7: Account can not be liquidated when price fall by 99%.

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

Found by

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

Issue M-8: `emergencyClosePosition()` is missing the `whenPaused` modifier

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/60

Found by

KingNFT, roguereddwarf, Diana

Summary

The `emergencyClosePosition()` is designed to be called when the contract is paused, as the call may cause users' financial loss. But it is missing to append the `whenPaused` modifier. By calling it, users may suffer unwarranted losses when the contract is not paused.

Vulnerability Detail

```
File: contracts\Vault\BaseVault.sol
221:     /**
222:      * @dev Function allowing a user to automatically close their position.
223:      * Note that this function is available even when the contract is
↪   paused.
224:      * Note that since this function does not call updateReward, it should
↪   only be used when the contract is paused.
225:      *
226:      */
227:     function emergencyClosePosition() external { // @audit not paused
228:         _modifyPosition(msg.sender, userDetails[msg.sender].collateral,
↪   userDetails[msg.sender].debt, false, false);
229:     }
```

Impact

Users may suffer unexpected losses when the contract is not paused

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/Vault/BaseVault.sol#L227

Tool used

Manual Review

Recommendation

append the `whenPaused` modifier.

Discussion

iHarishKumar

https://github.com/protokol/taurus-contracts/pull/110

Issue M-9: 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

Issue M-10: 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

roguereddwarf

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

Issue M-11: Liquidation bot is vulnerable to sandwich attack

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/19

Found by

Chinmay, yixxas, cducrest-brainbot, Ityu

Summary

Bot attempts to liquidate accounts that are no longer healthy, but hardcodes `_minExchangeRate` to be 0. `_minExchangeRate` is used as slippage check for collateral received when a position is liquidated hence it is an important protection from sandwich attacks.

Vulnerability Detail

Liquidation bot is used to constantly scan for accounts that are no longer healthy via offchain. Because the liquidation bot will liquidate a position and accept any exchange rate, an adversary can perform a sandwich attack.

Adversary can inflate the price of collateral before this liquidate transaction, and force the liquidation bot to receive a much lower amount of collateral, and earning the difference.

Impact

Adversary can perform a sandwich attack, arbitraging liquidation process at the cost of liquidation bot, resulting in it receiving less collateral than it should for liquidating a position.

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/LiquidationBot/LiquidationBot.sol#L89-L106

Tool used

Manual Review

Recommendation

Consider setting a reasonable `_minExchangeRate` for when calling liquidate via the bot, instead of setting it perpetually at 0.

Discussion

Sierraescape

Currently this isn't an issue since the GlpOracle can't reasonably be manipulated, but this seems like a great sanity check, especially for future vaults which may also use the LiquidationBot.

iHarishKumar

https://github.com/protokol/taurus-contracts/pull/112

Issue M-12: TAU.sol: Governance address cannot be changed

Source: https://github.com/sherlock-audit/2023-03-taurus-judging/issues/14

Found by

roguereddwarf

Summary

The `TAU` contract allows to set the `governance` address in its constructor. This is the address that is able to call `setMintLimit` function. The function is very important for the protocol because it manages which vaults can mint how much TAU.

The issue is that the `governance` address cannot be changed.

Vulnerability Detail

According to the contest page, the governance address may have to change:

> Governance. Entirely trusted. This role will be initially granted to the multisig.

Governance is initially the multisig but it should be possible to change it later on.

Impact

The `governance` of the `TAU` contract cannot be changed. This makes it impossible to change governance of the protocol since `TAU` is probably the most important part of all in the protocol

Code Snippet

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/TAU.sol#L17-L19

https://github.com/sherlock-audit/2023-03-taurus/blob/main/taurus-contracts/contracts/TAU.sol#L28-L33

Tool used

Manual Review

Recommendation

Implement functionality such that `governance` can transfer its role to another address. Maybe make use of the `controller` that the Vault makes use of for access controls such that all roles are managed in one place.

Discussion

iHarishKumar

https://github.com/protokol/taurus-contracts/pull/117