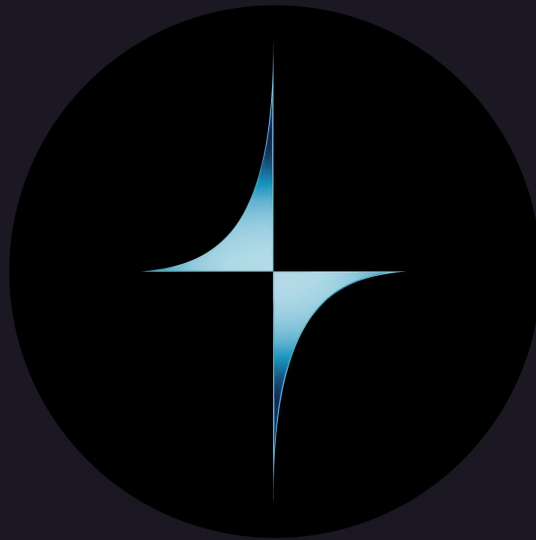




Security Review For Tangent



Public Best Efforts Audit Contest Prepared For: **Tangent**
Lead Security Expert: **Obsidian**
Date Audited: **August 28 - September 11, 2025**

Introduction

USG is a debt collateralized stablecoin backed by productive LPs and yield-bearing assets from blue chip DeFi protocols. The focus of the audit is our lending mechanism, dynamic interest rate model, LP oracle, integrations with Curve, Convex, and Pendle, as well as reward streaming and zapping functionalities.

Scope

Repository: [Tangent-labs/tangent-contracts](https://github.com/tangent-labs/tangent-contracts)

Audited Commit: [d0fe823accbfed7adc27eb18824dd2d401741ba5](https://github.com/tangent-labs/tangent-contracts/commit/d0fe823accbfed7adc27eb18824dd2d401741ba5)

Final Commit: [99803b6afd55ebacf53a80cae2fbed68f8c1fbec](https://github.com/tangent-labs/tangent-contracts/commit/99803b6afd55ebacf53a80cae2fbed68f8c1fbec)

Files:

- [src/USG/Market/abstract/Collateral.sol](#)
- [src/USG/Market/abstract/DebtIR.sol](#)
- [src/USG/Market/abstract/MarketCore.sol](#)
- [src/USG/Market/abstract/MarketExternalActions.sol](#)
- [src/USG/Market/abstract/PauseSettings.sol](#)
- [src/USG/Market/BasicERC20Market.sol](#)
- [src/USG/Market/Convex/ConvexCrvLPMarket.sol](#)
- [src/USG/Market/Convex/ConvexFxnLPMarket.sol](#)
- [src/USG/Oracles/CurveLP/OracleCryptoSwap.sol](#)
- [src/USG/Oracles/CurveLP/OracleDuoPoolStable.sol](#)
- [src/USG/Oracles/OracleBase.sol](#)
- [src/USG/Oracles/Pendle/OraclePendlePT.sol](#)
- [src/USG/Oracles/Token/ChainlinkAggregatorWrapper.sol](#)
- [src/USG/Oracles/Token/OracleCoinFromCurveLP.sol](#)
- [src/USG/Oracles/Token/OracleERC4626.sol](#)
- [src/USG/Routers/PendlePTRouter.sol](#)
- [src/USG/Tokens/TAN.sol](#)
- [src/USG/Tokens/USG.sol](#)
- [src/USG/Tokens/VsTAN.sol](#)
- [src/USG/Tokens/WStable.sol](#)
- [src/USG/Utilities/abstract/LightOwnable.sol](#)

- src/USG/Utilities/abstract/LightReentrancyGuardTransient.sol
- src/USG/Utilities/abstract/ZappingUtil.sol
- src/USG/Utilities/ControlTower.sol
- src/USG/Utilities/IRCalculator.sol
- src/USG/Utilities/MarketCreator.sol
- src/USG/Utilities/Migratoor.sol
- src/USG/Utilities/RewardAccumulator.sol
- src/USG/Utilities/ZappingProxy.sol

Final Commit Hash

99803b6afd55ebacf53a80cae2fbed68f8c1fbec

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- 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.

Issues Found

High	Medium
2	14

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0xAdwa
0xDelvine
0xFlare
0xHexed
0xImmortan
0xRaz
0xRstStn
0xShoonya
0xSolus
0xc0ffEE
0xd4ps
0xdoichantran
0xodus
0xpetern
0xsh
0xxAristos
3rdeye
7
8olidity
AMOW
Artur
Audinarey
AuditorPraise
BADROBINX
BAdal-Sharma-09
Berring
Bigsam
Bluedragon
Brene
ChaosSR
Cybrid
DharkArtz
Elroi
ExtraCaterpillar
HeckerTrieuTien
InquisitorScythe
JeRRy0422
JuggerNaut
Mishkat6451
Nyxx
Obsidian
Orhukl

Oxsadeeq
ParthMandale
Pexy
Pianist
PratRed
Pro_King
Ratt13snak3
Sa1ntRobi
SiddiqX7860
Sneks
Suryaa__
Synthrax
Tigerfrake
Uddercover
VCeb
Vesko210
VictorUbi
X0sauce
Xmanuel
Yaneca_b
Ziusz
air_0x
algiz
almurhasan
aman
ami
auditorshambu
axelot
bam0x7
befree3x
boredpukar
ck
coin2own
covey0x07
curly
deadmanwalking
edger
ezsia
gh0xt
glitch-Hunter
gneiss
greekfreakxyz

heavyw8t
holtzzx
illoy_sci
itsRavin
jayjoshix
joicygiore
khaye26
kimnoic
lucky-gru
magickenn
maigadoh
makarov
maxim371
merlin
mgf15
molaratai
newspacexyz
oxelmiguel
prosper
redtrama
roshark
sakibcy
shieldrey
silver_eth
sl1
sourav_DEV
theboiledcorn
theholymarvycodes
tobi0x18
tyuuu
udo
v1c7
vivekd
web3made
weblogicctf
wickie
x15
xKeywordx
yaioxy
yoooo
zh1xlan1221

Issue H-1: Expired PT doesn't always redeems 1:1 for underlying token

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/51>

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xShoonya, 0xdoichantran, ExtraCaterpillar, Orhukl, PratRed, covey0x07, greekfreakxyz, newspacexyz, theboiledcorn, tobi0x18, tyuuu, weblogicctf, yoooo

Summary

The `OraclePendlePT` contract assumes that once a Pendle Principal Token (PT) expires, its value becomes exactly 1:1 with the underlying asset. This is based on the assumption that 1 PT can be redeemed for exactly 1 unit of the underlying via the SY token mechanism. However, in practice, the redemption path of expired PT often involves external DEX swaps, which may be subject to MEV, low liquidity, or price impact – especially during volatile market conditions. As a result, the actual value of 1 PT at expiry may be less than 1 underlying token, leading to oracle overvaluation.

Root Cause

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Oracles/Pendle/OraclePendlePT.sol#L41-L43> The contract assumes a guaranteed 1:1 redemption of PT to underlying asset post-expiry, without validating the execution path's market conditions or slippage.

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

.

Impact

Mispriced expired PTs will cause incorrect borrowing position for users and protocol will lose funds

PoC

No response

Mitigation

No response

Issue H-2: Caller supplied ControlTower lets anyone be the migrator

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/61>

Found by

0xAdwa, 0xImmortan, 0xRaz, 0xRstStn, 0xc0ffEE, 0xd4ps, 0xodus, 0xxAristos, 7, Artur, BADROBINX, Cybrid, HeckerTrieuTien, InquisitorScythe, JeRRy0422, Obsidian, Orhukl, Oxsadeeq, Pexy, Rattl3snak3, Sneks, Synthrax, Uddercover, X0sauce, Ziusz, air_0x, axelot, bam0x7, boredpukar, covey0x07, edger, gh0xt, gneiss, greekfreakxyz, illoy_sci, jayjoshix, lucky-gru, maigadoh, makarov, oxelmiguel, prosper, redtrama, sakibcy, slI, theboiledcorn, tobi0x18, web3made, x15, xKeywordx, yaioxy

Summary

The migration and reentrancy functions verify `isPositionMigrator(msg.sender)` against a caller-provided `IControlTower` parameter instead of the market's trusted `controlTower` state. An attacker can pass a fake `ControlTower` that always returns true, then use `migrateFrom/migrateTo` to siphon a victim's collateral to an attacker receiver, shove debt onto others, and grief by toggling the reentrancy lock

Root Cause

The contract delegates the role check to a caller-supplied parameter instead of its own trusted state. In `_verifySenderMigrator`, the authority being queried is the external argument `_controlTower`, which the caller controls

```
// MarketCore.sol
function _verifySenderMigrator(IControlTower _controlTower) internal view {
    // @ Untrusted authority: the caller picks _controlTower
    require(_controlTower.isPositionMigrator(msg.sender), NotAMigrator());
}
```

Entry points then accept that same `_controlTower` and pass it through

```
// MarketExternalActions.sol (simplified)
function migrateFrom(
    IControlTower _controlTower,          // @ user-supplied
    address account,
    uint256 collatToRemove,
    uint256 debtToRemove,
    uint256 debtToRepay,
    address receiver
) external /* nonReentrant etc */ {
```

```

    _verifySenderMigrator(_controlTower); // @ checks the untrusted param
    // ... proceeds with collateral/debt moves, eventually sending to `receiver`
}

function migrateTo(
    IControlTower _controlTower, // @ user-supplied
    address account,
    uint256 collatToAdd,
    uint256 debtToAdd
) external {
    _verifySenderMigrator(_controlTower); // @ same issue
    // ... mutates another account's position
}

function reeentrancyOn(IControlTower _controlTower) external {
    _verifySenderMigrator(_controlTower); // @ lets attacker flip lock
    // ... set lock = true
}

```

Conceptually, the market already has a trusted authority in storage

```
IControlTower public controlTower; // @ canonical, trusted authority for roles
```

But the checks ignore it and instead trust whatever the external caller passes in.

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

1. Attacker deploys FakeControlTower implementing isPositionMigrator(address) → true.
2. Calls market.migrateFrom(fakeCT, victim, collatToRemove, debtToRemove, debtToRepay, receiver=attacker).
3. Function proceeds, reducing victim's collateral (and optionally reshaping debt), ultimately transferring collateral out via the post-hooks to receiver.
4. Optionally call migrateTo(fakeCT, anotherVictim, collatToAdd, debtToAdd) to push debt onto another account.

Impact

Any caller can masquerade as an authorized migrator by passing a fake `IControlTower` that always approves them. That lets the attacker invoke `migrateFrom` to siphon a victim's collateral to an attacker-controlled receiver, potentially draining positions entirely in a single transaction.

Using `migrateTo`, the attacker can shove arbitrary debt onto other accounts, forcing liquidations, ruining health factors, and creating broad griefing/insolvency risk for the market. They can also flip the market's reentrancy lock via `reeantrancyOn/Off` to disrupt other users' actions during execution, enabling denial-of-service style griefing.

There are no meaningful preconditions beyond deploying a permissive `ControlTower` and calling the functions, so the blast radius includes any account with collateral on any affected market.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: <https://github.com/Tangent-labs/tangent-contracts/commit/925e46fafbd3465feff24bfc4badfb64f3fc6a42>

Issue M-1: Incorrect assumption of fixed 1e18 scaling in Pendle PT Oracle Rate

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/40>

Found by

0xSolus, 0xdoichantran, Artur, AuditorPraise, BADROBINX, Bluedragon, Brene, ExtraCaterpillar, Oxsadeeq, Pro_King, Suryaa__, Tigerfrake, X0sauce, ck, covey0x07, greekfreakxyz, lucky-gru, magickenn, newspacexyz, roshark, shieldrey, tobi0x18, vlc7

Summary

The `OraclePendlePT` contract assumes that the `getPtToSyRate()` function from Pendle's oracle always returns a rate scaled to 1e18. However, some Pendle markets (e.g., `0x83916356556f51dcBcB226202c3efeEfc88d5eaA`) return values with larger decimal scaling, which causes incorrect PT pricing.

Root Cause

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Oracles/Pendle/OraclePendlePT.sol#L45> `getPtToSyRate()` function will return `ptRate` in larger decimals for some markets.

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

For market: `0x83916356556f51dcBcB226202c3efeEfc88d5eaA` <https://etherscan.io/address/0x9a9Fa8338dd5E5B2188006f1Cd2Ef26d921650C2#readProxyContract#F7> [`getPtToSyRate(address,uint32)` method Response] `uint256` : `999837196509292391919823695` The decimals is not 1e18.

Impact

Incorrect `ptRate` will cause the incorrect price returned by the oracle. This will be critical for contracts such as `Collateral` couldn't work properly. Due to this, liquidation won't work and protocol will lose funds.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/121>

Issue M-2: Incorrect assumption that one (1) Pendle SY token equals one (1) Yield Token in oracle pricing

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/50>

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

OxSolus, BADROBINX, Bluedragon, Mishkat6451, Orhukl, Suryaa__, X0sauce, algiz, deadmanwalking, greekfreakxyz, magickenn, newspacexyz, vlc7

Summary

The `OraclePendlePT` contract incorrectly assumes that 1 Pendle Standard Yield (SY) token is always equal in value to 1 unit of the underlying Yield Token. This assumption is embedded in the following pricing formula:

```
return (oracle.getPtToSyRate(address(_params.pendleMarket),  
    ↳ uint32(_params.duration)) * underlyingPrice) / 1e18;
```

However, in real Pendle markets, the SY token can deviate significantly from a 1:1 peg with the Yield Token, depending on market conditions, liquidity, and accrued yield. By treating SY as if it is always equal to the Yield Token, the oracle overstates the value of the PT token, introducing systemic mispricing. Pendle's SY.redeem function showing that slippage might occur during the exchange, and thus 1 SY == 1 Yield Token does not always hold. <https://github.com/pendle-finance/pendle-core-v2-public/blob/46d13ce4168e8c5ad9e5641dd6380fea69e48490/contracts/interfaces/IStandardizedYield.sol#L87>

Root Cause

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Oracles/Pendle/OraclePendlePT.sol#L45> A hardcoded pricing formula that assumes SY = underlying asset, without checking or incorporating the actual market price of the SY token.

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

.

Impact

Inflated oracle price will cause liquidation revert and protocol broken. Users can borrow more than expected and never liquidated affecting the protocol's solvency and increasing the risk of bad debt. Thus this will lead to financial loss or insolvency to the protocol.

PoC

No response

Mitigation

No response

Issue M-3: USG peg assumption in on-chain safety checks incorrect liquidation/insolvency decisions when USG depegs

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/102>

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

AMOW, BADROBINX, X0sauce, Xmanuel, covey0x07, oxelmiguel, sl1, tobi0x18

Summary `MarketCore` computes health, LTV and liquidation eligibility using collateral USD price but does not convert USG-denominated debt to USD (it assumes $1 \text{ USG} == \$1$), so a USG depeg causes incorrect safety checks that can produce silent insolvency or unfair liquidations. The code compares USD-valued collateral (via a collateral oracle) against USG token units without converting the USG amount into USD using a USG price oracle. This is a unit mismatch: collateral is valued in USD, debt is denominated in USG, yet the comparison assumes $1 \text{ USG} == \$1$ on-chain.

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Market/abstract/Collateral.sol#L192>

The function returns: <https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Market/abstract/Collateral.sol#L194> returns USD price ($1e18$), but `userDebt_` is raw USG token units; there is no multiplication by USG price to convert debt \rightarrow USD.

Impacts Silent insolvency Incorrect liquidation / incentive mismatch Operational risk due to off-chain keepers

Recommendation Convert debt to USD for safety checks

Issue M-4: Lack of USDT support due to use of transfer

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/127>

Found by

0xFlare, 0xShoonya, 0xSolus, 0xpetern, 0xxAristos, Audinarey, BAdal-Sharma-09, ChaosSR, Cybrid, Elroi, ExtraCaterpillar, JuggerNaut, Obsidian, SaIntRobi, VCeb, X0sauce, Yaneca_b, algiz, axelot, covey0x07, glitch-Hunter, heavyw8t, khaye26, merlin, mgf15, molaratai, redtrama, tobi0x18, wickie, yoooo, zh1xlan1221

Vulnerability Details

In README, it is stated that any token can be used to zap into output tokens that can be use to supply to market for any zapping actions (zapDeposit, zapRepayAndWithdraw, zapRepay, zapLeverage).

It is observed that any leftover tokenIn supplied by the user in exchange for the tokenOut of choice will be send to the protocol's treasury.

```
...
    if (msg.value == 0) {
        balanceTokenInLeft = tokenIn.balanceOf(address(this));
        if (0 != balanceTokenInLeft) {
@>            tokenIn.transfer(controlTower.feeTreasury(), balanceTokenInLeft);
        }
    }
```

However for tokens like USDT, This means it is unable to support USDT due to the use of transfer, as these function do not return a boolean value. This means in the event that there are leftover amounts of USDT during calls to ZappingProxy.zapProxy, the call will consistently fail.

This results in consistent DOS for users who attempt to conduct Market actions using USDT.

Root Cause

Use of transfer function prevent USDT usage.

Mitigation

Use safeTransfer from SafeERC20 library.

LOC

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Utilities/ZappingProxy.sol#L66>

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/114>

Issue M-5: ZappingProxy cannot receive ETH refunds resulting in failed zaps

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/150>

Found by

0xShoonya, covey0x07, deadmanwalking, holtzzx, jayjoshix, kimnoic, tobi0x18

Summary

ZappingProxy forwards `msg.value` to routers but has no payable receive/fallback, so router ETH refunds to the proxy will revert. This causes failed zaps for ETH paths that expect refunds.

Root Cause

When `zapProxy` is called with `tokenIn == CHAIN_COIN` and a positive `msg.value`, it forwards ETH to the router: [ZappingProxy.sol#L39-L76](#)

```
uint256 bal = tokenOut.balanceOf(receiver);

(bool isRouterCallSuccess, bytes memory data) = router.call{value:
    ↪ msg.value}(zap.routerCall);

require(isRouterCallSuccess, ZapCallError(data));
```

If there is unspent ETH, router refund unspent ETH to caller/`msg.sender`.

This is the logic present in `zapProxy` to sweep any ETH balance to the treasury:

```
} else {
    balanceTokenInLeft = address(this).balance;
    if (0 != balanceTokenInLeft) {
        payable(controlTower.feeTreasury()).transfer(balanceTokenInLeft);
    }
}
```

But there is no payable `receive()` or `fallback()` in `ZappingProxy`, so ETH refunds to the proxy will revert.

Internal pre-conditions

External pre-conditions

.

Attack Scenario

.

Impact

Because ZappingProxy lacks a payable receive()/fallback(), the refund reverts, causing the entire zap to revert.

Mitigation

Add a payable ETH acceptance path (receive/fallback) so router refunds don't revert.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/116>

Issue M-6: No slippage check for liquidators when they burn USG from their account without Swapping first.

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/178>

Found by

3rdeye, BADROBINX, Bigsam, Brene, Pro_King, Tigerfrake, VCeb, X0sauce, tobi0x18

Summary

Loss of funds by the liquidator when they liquidate a position, because of no slippage check during the liquidator's USG balance usage.

Root Cause

```
function _preLiquidate(address account) internal returns (uint256, uint256,
↳ uint256, uint256) {
    uint256 newDebtIndex = _checkpointIR();
    uint256 userDebtShares_ = userDebtShares[account];
```

```
@audit>>         return (newDebtIndex, collateralBalances[account], userDebtShares_,
↳ _convertToAmount(userDebtShares_, newDebtIndex));
}
```

```
function _liquidate(LiquidateInput memory liquidateInput, ZapStruct calldata
↳ liquidateCall) internal returns (uint256, uint256, uint256, uint256) {
    uint256 collatAmountToLiquidate = liquidateInput.collatToLiquidate;
```

```
_verifyCollatInputNotZero(liquidateInput.collatToLiquidate);
```

```
uint256 debtSharesToRemove;
uint256 USGToRepay;
```

```
// Liquidate all
```

```
@audit>>         if (collatAmountToLiquidate >= liquidateInput._collateralBalance) {
```

```
@audit>>             collatAmountToLiquidate = liquidateInput._collateralBalance;
                USGToRepay = liquidateInput.userDebt;
                debtSharesToRemove = liquidateInput._userDebtShares;
        }
```

```

        // Liquidate partial
        else {

@audit>>            USGToRepay = (collatAmountToLiquidate *
↪ liquidateInput.userDebt) / liquidateInput._collateralBalance;
@audit>>            debtSharesToRemove = _convertToShares(USGToRepay,
↪ liquidateInput.newDebtIndex);

            // Ensure that the remaining debt is bigger than a minimum to leave
            ↪ profitable liquidation
            _verifyMinimumDebt(liquidateInput.userDebt - USGToRepay);
        }

        uint256 newUserDebtShares = liquidateInput._userDebtShares -
        ↪ debtSharesToRemove;
        // Modify the collateral balance, the user debt and the total debt
        _updateCollatAndDebts(
            liquidateInput.account,
            liquidateInput._collateralBalance - collatAmountToLiquidate,
            liquidateInput._totalCollateral - collatAmountToLiquidate,
            newUserDebtShares,
            liquidateInput._totalDebtShares - debtSharesToRemove
        );

        uint256 fee = _mulDiv(USGToRepay, liquidationFee, DENOMINATOR);
        _postLiquidate(collatAmountToLiquidate, USGToRepay + fee,
        ↪ liquidateInput.minUSGOut, liquidateCall);

        if (fee != 0) {
            _mintUSG(usg, controlTower.feeTreasury(), fee);
        }

        return (collatAmountToLiquidate, USGToRepay, fee, newUserDebtShares);
    }

```

When a liquidator liquidates with an account, they state the amount of collateral to seize and the minimum USG to receive. But this is only enforced when using ZAP (COLLATERAL to USG).

But Liquidators who liquidate with their balance cannot enforce the maximum USG to be burned from their account for a token valuation.

Collateral worth \$990 debt worth \$960 plus a fee of 20 Liquidator sets \$981 as minUSG

At the point of liquidation execution, call to execution price, the valuation reads Collateral \$978 and debt plus fee \$980, slippage not enforced with Zap path, this would have failed.

```

/**
 * @dev Internal function called at the end of 'liquidate' and 'selfLiquidate'
 ↪ external function.

```

```

*      Withdraw the collateral from the underlying protocol
*      Transfer the collateral to the caller or to the zapping proxy
*      When the collateral is sent to the zapping proxy, the
↳ 'liquidationCall' handles the selling of the collateral
* @param collateralAmountToLiquidate Amount of collateral to sell during the
↳ liquidation
* @param USGToBurn Amount of USG to burn from the sender
* @param minUSGOut Slippage, Minimum amount of USG to receive
↳ after the selling of the collateral
* @param liquidationCall Contains address and bytes of the contract
↳ selling the collateral for USG
*/
function _postLiquidate(uint256 collatAmountToLiquidate, uint256 USGToBurn,
↳ uint256 minUSGOut, ZapStruct calldata liquidationCall) internal {
    IZappingProxy _zappingProxy = zappingProxy;
    // Withdraw the collateral from the underlying protocol if needed and
    // Transfer it to the caller when there is no liquidator passed in the
    ↳ parameter
    // If a liquidator is passed, we send the collateral to the Zapping Proxy
    ↳ that will handle the selling of the collateral.
    _transferCollateralWithdraw(liquidationCall.router != address(0) ?
    ↳ address(_zappingProxy) : msg.sender, collatAmountToLiquidate);
    // When the liquidator is not zero, it allows the LiquidatorProxy to
    ↳ receive the collateral.
    // Then, if needed, the liquidator will allow the custom Liquidator to sell
    ↳ the collateral for USG in the same transaction.
    if (liquidationCall.router != address(0)) {

@audit>>         _zappingProxy.zapProxy(collatToken, usg, minUSGOut, msg.sender,
↳ liquidationCall);
    }

    // Burns USG from the sender.
    // The debt has to be on the caller of the transaction.
    // In case a liquidator is passed in parameter, it needs to send it back to
    ↳ the sender of the tx.

@audit>>         _burnUSG(msg.sender, USGToBurn);

    }

```

Zapping ensures we check collateral valuation after swapping against minOUT, but when using Msg. Sender, we fail to do the same.e

```

* @notice Liquidate a position that has a health ratio < 1.
* @dev      Two liquidation modes are possible
*           - Buy USG with a flashloan, repay the debt, get the collateral and do
↳ whatever you want with it.

```

```

        - Selling the collateral for USG directly through ZappingProxy by
        ↳ providing a route to repay the debt and keeping the difference in
        ↳ USG
    * @param account            Account of the position to liquidate
    * @param collatToLiquidate  Amount of collateral to liquidate from the position.
    * @param minUSGOut          Minimum amount of USG to receive on the sale of the
    ↳ collateral.
    * @param liquidationCall    Contract and data allowing to sell the collateral
    ↳ for USG.
    */
function liquidate(address account, uint256 collatToLiquidate, uint256 minUSGOut,
↳ ZapStruct calldata liquidationCall) external nonReentrant
↳ updateRewards(account) {
    (uint256 newDebtIndex, uint256 collatBalance, uint256 _userDebtShares, uint256
    ↳ userDebt_) = _preLiquidate(account);
    // Can liquidate only if the health ratio is below 1
    require(_healthRatio(userDebt_, collatBalance, true) < 1 ether,
    ↳ NotLiquidablePosition());

    (uint256 collatLiquidated, uint256 debtRepaid, uint256 fee, uint256
    ↳ newUserDebtShares) = _liquidate(
        LiquidateInput({
            account: account,
            collatToLiquidate: collatToLiquidate,
            minUSGOut: minUSGOut,
            newDebtIndex: newDebtIndex,
            _collateralBalance: collatBalance,
            _totalCollateral: totalCollateral,
            _userDebtShares: _userDebtShares,
            _totalDebtShares: totalDebtShares,
            userDebt: userDebt_
        }),
        liquidationCall
    );

    emit Liquidate(account, debtRepaid, newUserDebtShares, fee, collatLiquidated,
    ↳ liquidationCall.router);
}

```

Internal Pre-conditions

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Market/abstract/MarketCore.sol#L408-L450>

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Market/abstract/MarketCore.sol#L452-L478>

External Pre-conditions

.....

Attack Path

Liquidation lacks protection for the Liquidator when Zap is not used.

Impact

The Liquidator can receive less Collateral in the valuation for their USG token for debt and fees.

PoC

No response

Mitigation

Allow slippage check against the amount the liquidator is willing to burn from their USG for that collateral, ensuring that liquidators can specify the max amount to burn plus the fee for the collateral taken.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/117>

Issue M-7: Incorrect calls and Enforcements during Migration To a market.

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/180>

Found by

7, Artur, Bigsam, Brene, Tigerfrake, aman

Summary

Migration allows users to move collateral and debt, or collateral alone or debt alone. `_migrateTo` unconditionally enforces deposit and borrow pause checks and always calls `_postDeposit`, even when the migration only moves debt. This creates an unnecessary DoS on migration in legitimate cases and can block users from rescuing positions during critical moments.

Root Cause

```
function _migrateTo(IControlTower _controlTower, address account, uint256
    ↳ collatToAdd, uint256 debtToAdd) internal returns (uint256) {
    _verifySenderMigrator(_controlTower);

@audit>>         _verifyIsDepositNotPaused();           // check if we are adding
    ↳ collateral.....
@audit>>         _verifyIsBorrowNotPaused();             // check only if we are
    ↳ borrowing when we are not why check?

    uint256 debtIndex = _checkpointIR();

    uint256 uDebtShares = userDebtShares[account];
    uint256 newUserDebt = _convertToAmount(uDebtShares, debtIndex) + debtToAdd;
    uint256 newCollatBalance = collateralBalances[account] + collatToAdd;
    uint256 sharesToAdd = _convertToShares(debtToAdd, debtIndex);
    uint256 newTotalDebtShares = totalDebtShares + sharesToAdd;
    uint256 newUserDebtShares = uDebtShares + sharesToAdd;

    // We check both condition on minimum loan and maxLTV
    // Dont need to check them when the new user debt is equal to 0
    if (newUserDebt != 0) {
        _verifyMinimumDebt(newUserDebt);
        _verifyMaxLTV(newCollatBalance, newUserDebt, false);
        _verifyDebtCap(badDebt, newTotalDebtShares, debtIndex);
    }
```

```

_updateCollatAndDebts(account, newCollatBalance, totalCollateral +
↳ collatToAdd, newUserDebtShares, newTotalDebtShares);

```

```

@audit>>         _postDeposit(collatToken);    // Only when deposit occurs

        return newUserDebtShares;
    }

```

```

@audit>>         /// @notice Pauses new deposits on the market
@audit>>         bool public isDepositPaused;

@audit>>         /// @notice Pauses new borrows on the market
@audit>>         bool public isBorrowPaused;

```

isDepositPaused is specifically used to Prevent New deposit to the Market.
isBorrowPaused is specifically used to Prevent New borrow action in the Market.

```

/**
 *
 @audit>>         * @dev Hook after deposit to allow extended logic such as staking the
↳ collateral in an underlying protocol.
@audit>>         *         When not override, does nothing. Otherwise, refers to the
↳ overriding implementation.
@audit>>         * @param _collatToken Collateral token being deposited.
        */

@audit>>         function _postDeposit(IERC20 _collatToken) internal virtual {}

```

_postDeposit is specifically call whenever a new deposit is made after a deposit to allow extended logic.

Migration incorrectly calls above 3 functions regardless of the action in play

The correct flow

1. Coll > 0 and debt > 0 . call all . Correct
2. Coll > 0 and debt == 0 . check new deposit is allowed and calls post-deposit ...
3. Coll == 0 and debt > 0 . check if new borrow /debt transferred is allowed

Migration 2 and 3 fails even in ideal and correct state.

This strictly enforces SPECIFIC SPECIAL calls that are individual deposit and borrow calls flow

```

/**
 * @dev Internal function called during 'deposit' external function.
 * @param _for Address of the user/position receiving collateral.
 * @param amountDeposited Amount of collateral deposited.
 */
function _deposit(address _for, uint256 amountDeposited, IERC20 _collatToken)
↳ internal {

@audit>> // Cannot deposit on a market with paused deposits
@audit>> _verifyIsDepositNotPaused();

@audit>> // Cannot deposit 0
@audit>> _verifyCollatInputNotZero(amountDeposited);

// Checkpoint the IR and indexes
_checkpointIR();
// Increase collateral balance of the position and update total debt
_updateCollateral(_for, collateralBalances[_for] + amountDeposited,
↳ totalCollateral + amountDeposited);

_postDeposit(_collatToken);
}

```

```

/**
 * @dev Internal function called during 'borrow', 'depositAndBorrow' and
↳ 'leverage' external functions.
 * @param receiver Address receiving the borrowed USG.
 * @param USGToBorrow Amount of USG to borrow.
 * @param collatAmount Amount of collateral owned by borrower.
 * @param isLeverage Whether this borrow is part of a leverage transaction.
 * @return Updated user debt shares after borrow.
 * @return Updated total debt shares for the market.
 */
function _borrow(address receiver, uint256 USGToBorrow, uint256 collatAmount,
↳ bool isLeverage) internal returns (uint256, uint256) {

@audit>> _verifyIsBorrowNotPaused();
@audit>> _verifyDebtInputNotZero(USGToBorrow);

uint256 newDebtIndex = _checkpointIR();

uint256 _userDebtShares = userDebtShares[msg.sender];

```

```

uint256 newUserDebt = USGToBorrow + _convertToAmount(_userDebtShares,
↳ newDebtIndex);

// Cache the new value in USG of the debt
uint256 newUserDebtShares = _convertToShares(USGToBorrow, newDebtIndex);

uint256 newTotalDebtShares = totalDebtShares + newUserDebtShares;

_verifyDebtCap(badDebt, newTotalDebtShares, newDebtIndex);

// Verify that newDebt is over the minimum loan
_verifyMinimumDebt(newUserDebt);

// Verify that the newDebt of the loan is not over the maximum borrowable
_verifyMaxLTV(collatAmount, newUserDebt, false);

// If it's a leverage transaction, USG is already minted before
if (!isLeverage) {
    // Mint USG to the receiver
    _mintUSG(usg, receiver, USGToBorrow);          // users can mint USG for
    ↳ free ..... unbacked..
}

return (_userDebtShares + newUserDebtShares, newTotalDebtShares);
}

```

Post deposit fails on 0 deposit..

```

function _postDeposit(IERC20 _collatToken) internal override {
    stakingProxyVault.deposit(_collatToken.balanceOf(address(this)), true);
}

function _postDeposit(IERC20 _collatToken) internal override {
    uint256 _pid = pid;

    // When pid = 0 / Means the Market is not yet linked to Convex
    if (_pid != 0) {
        CVX_BOOSTER.deposit(_pid, _collatToken.balanceOf(address(this)), true);
    }
}

//deposit crv for cvxCrv
//can locking immediately or defer locking to someone else by paying a fee.
//while users can choose to lock or defer, this is mostly in place so that
//the cvx reward contract isnt costly to claim rewards
function deposit(uint256 _amount, bool _lock, address _stakeAddress) public {

```

```

@audit>>>         require(_amount > 0, "!>0");

    //deposit lp tokens and stake
@audit>>>         function deposit(uint256 _pid, uint256 _amount, bool _stake) public
↳ returns(bool){
    require(!isShutdown,"shutdown");
    PoolInfo storage pool = poolInfo[_pid];
    require(pool.shutdown == false, "pool is closed");

    //send to proxy to stake
    address lptoken = pool.lptoken;
    IERC20(lptoken).safeTransferFrom(msg.sender, staker, _amount);

    //stake
    address gauge = pool.gauge;
    require(gauge != address(0), "!gauge setting");
    IStaker(staker).deposit(lptoken,gauge);

    //some gauges claim rewards when depositing, stash them in a seperate
    ↳ contract until next claim
    address stash = pool.stash;
    if(stash != address(0)){
        IStash(stash).stashRewards();
    }

    address token = pool.token;

@audit>>>         if(_stake){
    //mint here and send to rewards on user behalf
    ITokenMinter(token).mint(address(this),_amount);
    address rewardContract = pool.crvRewards;
    IERC20(token).safeApprove(rewardContract,0);
    IERC20(token).safeApprove(rewardContract,_amount);

@audit>>>         IRewards(rewardContract).stakeFor(msg.sender,_amount);

    function stakeFor(address _for, uint256 _amount)
        public
        updateReward(_for)
        returns(bool)
    {

@audit>>>         require(_amount > 0, 'RewardPool : Cannot stake 0');

```

Internal Pre-conditions

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Market/abstract/MarketCore.sol#L655-L679>

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Market/abstract/MarketCore.sol#L193-L196>

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Market/abstract/MarketCore.sol#L105-L114>

External Pre-conditions

.....

Attack Path

Improper check enforcement and handling of call-in migration in two specific edge cases.

Impact

This creates a Denial of service, leaving users at risk of being liquidated and losing their collateral.

This can prevent users from:

1. Moving debt away from a stressed market or into a safer one if they have excess collateral in another market (has 0 collateral migration fails) .
2. Adding collateral via migration to shore up health.

Result: increased liquidation risk and potential loss of user funds in volatile periods.

PoC

No response

Mitigation

To mitigate this, add a condition to each call and enforce only when necessary.

```
function _migrateTo(IControlTower _controlTower, address account, uint256
    ↪ collatToAdd, uint256 debtToAdd) internal returns (uint256) {
    _verifySenderMigrator(_controlTower);

if( collatToAdd > 0 ) {      _verifyIsDepositNotPaused();    }      // check if we
    ↪ are adding collateral.....
```

```
if (debtToAdd > 0 )    {      _verifyIsBorrowNotPaused();    }    // check only if
↳ we are borrowing when we are not, why check?

if (collatToAdd > 0)    {      _postDeposit(collatToken);    }    // Only when
↳ deposit occurs

    return newUserDebtShares;
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/115>

Issue M-8: WStable Contract Invariant Break in is-Saving=true Mode

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/293>

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

8olidity, Berring, DharkArtz, ExtraCaterpillar, Pro_King, SiddiqX7860, Yaneca_b, ami, befree3x, coin2own, covey0x07, curly, deadmanwalking, glitch-Hunter, itsRavin, jayjoshix, udo, vivekd

Summary

The `WStable` contract breaks the 1:1 invariant between vault shares and `WStable` tokens when using the `isSaving=true` mode. When yield accrues in the underlying ERC4626 vault between mint and burn operations, users permanently lose a portion of their vault shares, with the loss percentage directly proportional to the yield accrual.

Root Cause

The root cause is an asymmetric conversion logic between the `mint` and `burn` functions when `isSaving=true`, specifically when user shares have already accrued yield:

1. In `mint(amountIn, receiver, true)`:

- User provides vault shares (`amountIn`) that have already accrued yield
- Contract converts these yield-bearing shares to assets using `previewMint(amountIn)`
- User receives `WStable` tokens equal to the asset value (as asset value is more than shares already), capturing the accrued yield

2. In `burn(amount, receiver, true)`:

- User provides `WStable` tokens (`amount`) representing assets
- Contract converts assets back to shares using `previewWithdraw(amount)`
- User receives vault shares

When additional yield accrues between these operations, the share-to-asset ratio increases further, causing users to receive fewer shares than they initially deposited. This asymmetric conversion creates a permanent loss of shares for users, as the contract fails to account for the fact that the user's original shares already had accrued yield at the time of deposit.

Invariant from README: Users are always able to withdraw 1:1 their deposits NB : This one is not completely true. Because of rounding, it can even be false. The consequence of this is that the last user to withdraw its WStable will not be able to burn everything but everything - $\text{numberOfBurnSinceCreation} * 1\text{wei}$. This is mitigated because we'll mint in the WStable few K\$ to initiate the associated LP that we'll never remove. So we have a lot of room before this error occurs.

Internal Pre-conditions

- User must use the `isSaving=true` mode in both `mint` and `burn` functions
- The user's shares must have already accrued some yield before deposit
- Additional yield must accrue in the vault between the user's mint and burn operations

External Pre-conditions

Attack Path

1. User has vault shares that have already accrued yield (ratio 1:1.1)
 - Each share is worth 1.1 assets due to previous yield accrual
2. User deposits these yield-bearing vault shares using `mint(shares, user, true)`
 - User transfers X vault shares to the contract
 - Contract calculates asset value: $\text{assets} = \text{vault.previewMint}(X) = X * 1.1$
 - User receives $X * 1.1$ amount of WStable tokens
3. Additional yield accrues in the underlying vault (this happens naturally over time)
 - Share-to-asset ratio increases from 1:1.1 to 1:1.2 (where each share is now worth 1.2 assets)
4. User burns WStable tokens using `burn(assets, user, true)`
 - User burns $X * 1.1$ amount of WStable tokens
 - Contract calculates share value: $\text{shares} = \text{vault.previewWithdraw}(X * 1.1) = (X * 1.1) / 1.2 = X * 0.9167$
 - User receives $X * 0.9167$ amount of vault shares, which is less than the original X
5. Result: User has permanently lost vault shares (8.33% loss)

Impact

- **Loss of user funds:** Users permanently lose a portion of their vault shares proportional to the yield accrual
- In our test scenario, a user lost 8.33% of their shares when the share-to-asset ratio changed from 1:1.1 to 1:1.2
- This issue affects all users who use the `isSaving=true` mode, which is specifically designed for share-based operations

PoC

Place this file at **WStables** folder of test and run this command to run test `forge test --mt "test_InvariantBreak_Scenario" -vv`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "../contexts/MarketDeploymentContext.sol";

contract wUSDE_InvariantBreak is MarketDeploymentContext {
    IERC20 stable = AddrClassicERC20.USDe;
    IERC4626 saving = AddrERC4626.sUSDe;

    function test_InvariantBreak_Scenario() external {

        vm.startPrank(address(this));
        deal(address(stable), address(this), 10000 ether);
        stable.approve(address(saving), 10000 ether);
        saving.deposit(10000 ether, address(this)); // Get shares at 1:1 ratio
        vm.stopPrank();

        // Step 2: Accrue yield to create exactly 1 : 1.1 ratio
        uint256 currentShares = saving.totalSupply();
        uint256 targetAssetsFor11 = (currentShares * 11) / 10; // 1.1 ratio
        console.log("currentShares : ", currentShares);
        console.log("targetAssetsFor11 : ", targetAssetsFor11);
        deal(address(stable), address(saving), targetAssetsFor11);
        skip(1 days);

        uint256 ratio11 = saving.previewRedeem(1 ether);
        console.log("After first yield: 1 share = %s assets (exactly 1:1.1 ratio)",
            ↪ ratio11);

        // Verify we have approximately 1.1 ratio
        assertApproxEqRel(ratio11, 110000000000000000, 1e16, "Ratio should be
            ↪ approximately 1.1");
    }
}
```

```
// Step 3: Giving user 1000 vault shares (now at 1:1.1 ratio)
uint256 vaultShares = 1000 ether;
vm.startPrank(address(this));
saving.transfer(usr1, vaultShares);
vm.stopPrank();

uint256 initialShares = saving.balanceOf(usr1);
console.log("User gets 1000 vault shares (worth %s assets at 1:1.1 ratio)",
↳ saving.previewRedeem(initialShares));

// Step 4: User deposits 1000 vault shares using WStable contract
vm.startPrank(usr1);
saving.approve(address(wUSDE), vaultShares);
wUSDE.mint(vaultShares, usr1, true);
vm.stopPrank();

uint256 wStableReceived = wUSDE.balanceOf(usr1);
console.log("WStable tokens received: %s", wStableReceived);
console.log("Expected: 1000 * 1.1 = 1100 (exactly 11000000000000000000)");

assertApproxEqRel(wStableReceived, 11000000000000000000, 1e16, "Should
↳ receive approximately 1100 WStable tokens");

// Step 5: Accrue more yield to create 1:1.2 ratio
uint256 targetAssetsFor12 = (saving.totalSupply() * 12) / 10; // 1.2 ratio
console.log("targetAssetsFor12: ", targetAssetsFor12);
deal(address(stable), address(saving), targetAssetsFor12);
skip(1 days);

uint256 ratio12 = saving.previewRedeem(1 ether);
console.log("After second yield: 1 share = %s assets (exactly 1:1.2
↳ ratio)", ratio12);

assertApproxEqRel(ratio12, 12000000000000000000, 1e16, "Ratio should be
↳ approximately 1.2");

// Step 6: User burns WStable tokens using WStable contract
vm.startPrank(usr1);
wUSDE.burn(wStableReceived, usr1, true);
vm.stopPrank();

uint256 finalShares = saving.balanceOf(usr1);
uint256 sharesLost = initialShares - finalShares;
uint256 lossPercentage = (sharesLost * 100) / initialShares;

console.log("Final user vault shares: %s", finalShares);
console.log("Expected: 1100 / 1.2 = 916.67 (exactly
↳ 91666666666666666667)");
console.log("Shares lost: %s", sharesLost);
```

```

    console.log("Loss percentage: %s%%", lossPercentage);
    console.log("Expected loss: 83.33 shares (8.33%%)");

    // Verifying we get approximately 916.67 shares back
    assertApproxEqRel(finalShares, 91666666666666666667, 1e16, "Should receive
    ↪ approximately 916.67 shares");
    assertApproxEqRel(sharesLost, 8333333333333333333, 1e16, "Should lose
    ↪ approximately 83.33 shares");

    // The invariant is broken: user deposited more shares than they got back
    assertLt(finalShares, initialShares, "User should have fewer shares than
    ↪ they started with");
    assertGt(sharesLost, 0, "User should have lost shares due to invariant
    ↪ break");
    assertGt(lossPercentage, 5, "Loss should be significant (>5%) to clearly
    ↪ demonstrate the issue");

    console.log("INVARIANT BROKEN: Deposited %s shares, got back %s shares",
    ↪ initialShares, finalShares);
  }
}

```

Output of test:

```

Ran 1 test for test/unit/WStables/wUSDE_InvariantBreak.t.sol:wUSDE_InvariantBreak
[PASS] test_InvariantBreak_Scenario() (gas: 731009)
Logs:
  currentShares : 4404337623717191315390233149
  targetAssetsFor11 : 4844771386088910446929256463
  After first yield: 1 share = 10999999999999999999 assets (exactly 1:1.1 ratio)
  User gets 1000 vault shares (worth 10999999999999999999 assets at 1:1.1 ratio)
  WStable tokens received: 1100000000000000000000
  Expected: 1000 * 1.1 = 1100 (exactly 1100000000000000000000)
  targetAssetsFor12: 5285205148460629578468279778
  After second yield: 1 share = 11999999999999999999 assets (exactly 1:1.2 ratio)
  Final user vault shares: 91666666666666666667
  Expected: 1100 / 1.2 = 916.67 (exactly 91666666666666666667)
  Shares lost: 8333333333333333333
  Loss percentage: 8%
  Expected loss: 83.33 shares (8.33%)
  INVARIANT BROKEN: Deposited 100000000000000000000 shares, got back
  ↪ 91666666666666666667 shares

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.34s (5.24ms CPU time)

```

Mitigation

Share-Based Accounting: Track shares instead of assets when `isSaving=true`

Issue M-9: Users can steal accumulated rewards when totalCollateral becomes zero due to incomplete state updates

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/348>

Found by

0xHexed, 0xShoonya, BADROBINX, Bigsam, Brene, DharkArtz, HeckerTrieuTien, JeRRy0422, Mishkat6451, Sneks, Tigerfrake, Uddercover, Vesko210, almurhasan, auditorshambu, ck, ezsia, gh0xt, heavyw8t, illoy_sci, jayjoshix, kimnoic, maigadoh, maxim371, sll, sourav_DEV, theholymarvycodes, udo, vlc7, vivekd

Summary

When totalCollateral becomes zero, the `_updateReward()` function skips updating `userRewardPerTokenPaid` for new depositors, leaving it at 0 while `rewardPerTokenStored` retains previously accumulated values. This allows attackers to claim rewards that were already earned by previous users.

Root Cause

In `RewardAccumulator._updateReward()`, the entire reward update logic is skipped when `totalCollateral == 0`:

```
function _updateReward(address market, address account, uint256 collateralBalance,
    ↪ uint256 totalCollateral) internal {
    if (totalCollateral != 0) { // Skips when totalCollateral = 0
        // ... reward calculations and userRewardPerTokenPaid updates
    }
}
```

This leaves `userRewardPerTokenPaid[market][account][token]` at 0 instead of syncing it to the current `rewardPerTokenStored` value.

Internal Pre-conditions

1. `totalCollateral = 0`.
2. `rewardPerTokenStored` contains accumulated reward data from previous periods.

External Pre-conditions

None

Attack Path

1. `totalCollateral` needs to become zero.
2. Deposit minimal collateral (even 1 wei) to trigger `updateRewards(account)`.
3. Due to the bug, `userRewardPerTokenPaid[attacker][token]` remains 0 instead of being updated to current `rewardPerTokenStored`.
4. withdraw the deposited collateral.
5. During withdrawal, `_earned()` calculates: $(\text{balance} * (\text{rewardPerTokenStored} - 0)) / 1e18$ = full accumulated rewards.
6. Wait for protocol to accumulate sufficient reward token balance through new reward periods
7. Execute `claimSimple()` to claim the phantom rewards once contract has enough balance.

Impact

1. **Fund Loss:** Attacker claims rewards equivalent to previously accumulated `rewardPerTokenStored` values.
2. **Reward Pool Depletion:** Protocol reward balance gets drained by unearned claims.
3. **Legitimate other User Impact:** Other users cannot claim their actual earned rewards due to insufficient contract balance caused by Attacker claims

PoC

none

Mitigation

Always update `userRewardPerTokenPaid` to sync with current `rewardPerTokenStored`, even when `totalCollateral = 0`:

```
function _updateReward(address market, address account, uint256 collateralBalance,
    ↪ uint256 totalCollateral) internal {
    uint256 rewardLength = rewardTokens[market].length;

    for (uint256 i; i < rewardLength;) {
        IERC20 token = rewardTokens[market][i];
```

```

    if (totalCollateral != 0) {
        rewardData[market][token].rewardPerTokenStored =
            ↪ _rewardPerToken(market, token, totalCollateral);
        rewardData[market][token].lastUpdateTime =
            ↪ _lastTimeRewardApplicable(rewardData[market][token].periodFinish);
    }

    if (account != address(0)) {
        if (totalCollateral != 0) {
            rewards[market][account][token] = _earned(market, account, token,
                ↪ collateralBalance, totalCollateral);
        }
        // FIX: Always sync userRewardPerTokenPaid
        userRewardPerTokenPaid[market][account][token] =
            ↪ rewardData[market][token].rewardPerTokenStored;
    }

    unchecked { ++i; }
}
}

```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/120>

Issue M-10: There is no way for the admin to sweep rewards lost due to precision

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/432>

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

OxShoonya, BADROBINX, ExtraCaterpillar, Obsidian, Pianist, VictorUbi, ami, joicygiore, silver_eth, vlc7, wickie, yaioxy

Summary

When Rewards are being distributed, Precision will take place, and when it does, those tokens will be stuck in that contract with no way of being recovered, since there is no function to get them back

Root Cause

The contract has no function to recover tokens lost due to precision.

Attack Path

1. VsTAN::processRewards is called by the owner and adding 10000 USG
2. Resulting in 165 343 915 343 915 rewardRate.
3. As this will go on for 7 days, which means per Second' 16 534 391 534 391 534 is given.
4. Let's say there was only 1e18 supply and no update rewards have not been called for 7 days for simplicity, which calculates this

```
(((_lastTimeRewardApplicable(rewardData[_rewardToken].periodFinish) -  
↪ rewardData[_rewardToken].lastUpdateTime) * rewardData[_rewardToken].rewardRate  
↪ * 1e18) /  
totalSupplyVsTan)
```

```
(((((block.timestamp) - ((block.timestamp - 7 days))) * 16534391534391534) * 1e18 /  
1e18)
```

- Answer: 9999999999999999763200.. Loss = 236 800

5. Now this precision value gets bigger when it comes to low decimal tokens like USDC, which would cause rewardRate to be 16 534.

6. The loss for 6 decimal token is = 236 800.. which in 5 cycles is more than \$1.. Also happens in AccumulatorReward
7. With RewardAccumulator holding all rewards and updating of rewards, precision will happen, but those tokens won't be able to get recovered

Code

- <https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Utilities/RewardAccumulator.sol#L586C8-L590C10>

```
if (block.timestamp >= rData.periodFinish) {
    rData.rewardRate = rewardAmountStreamed / REWARDS_DURATION;
} else {
    rData.rewardRate = (rewardAmountStreamed + (rData.periodFinish -
    ↪ block.timestamp) * rData.rewardRate) / REWARDS_DURATION;
}
```

- <https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Tokens/VsTAN.sol#L556C6-L561C14>

```
if (timestamp >= rData.periodFinish) {
    rewardData[rewardToken].rewardRate = amount / ONE_WEEK;
} else {
    uint256 leftover = (rData.periodFinish - timestamp) * rData.rewardRate;
    rewardData[rewardToken].rewardRate = (amount + leftover) / ONE_WEEK;
}
```

Impact

The funds lost to precision cannot be recovered

Recommendation

Add a function that would be able to sweep Reward Tokens stuck in a contract

Issue M-11: WStable 1:1 exit path breaks with cooldown enabled ERC4626 vaults like sUSDe

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/633>

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

ExtraCaterpillar, aman, curly, greekfreakxyz, tobi0x18

Summary

The WStable contract has a flaw that makes it incompatible with a large class of ERC4626 vaults that implement withdrawal delays, cooldowns, or queues. For example, the planned deployment of wUSDE WStable contract with sUSDe (and any other similar vault) as the underlying ERC4626 saving account will prevent users from executing instant 1:1 withdrawals as intended. The sUSDe contract enforces a cooldown period (currently 604,800 seconds = 7 days) that causes all `withdraw()` calls to revert when the cooldown is enabled (which is the default state). This forces users to either receive sUSDe tokens instead of USDe (requiring additional unstaking steps and waiting periods) or be completely unable to withdraw their funds, breaking the core 1:1 withdrawal invariant and violating Sherlock guidelines regarding fund accessibility.

Root Cause

The WStable contract's `burn()` and `redeem()` functions attempt to call `savingAccount.withdraw()` when `isSaving = false`, but the sUSDe contract's `withdraw()` function includes an `ensureCooldownOff` modifier that reverts when cooldown is enabled:

Snippet from sUSDe contract on Etherscan:

```
function withdraw(uint256 assets, address receiver, address _owner)
    public
    virtual
    override
    @> ensureCooldownOff
    returns (uint256)
    {
        return super.withdraw(assets, receiver, _owner);
    }

    @> modifier ensureCooldownOff() {
        if (cooldownDuration != 0) revert OperationNotAllowed();
```

```
} _;
```

Current sUSDe contract state: `cooldownDuration` = 604800 -> 7 days. (from etherscan)

And here is the broken redeem and burn function on the WStable contract:

<https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Tokens/WStable.sol#L102-L105> <https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Tokens/WStable.sol#L80-L92>

Also a look at the deployment scripts verifies the intent to deploy the USDe and sUSDe pair: <https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/test/contexts/WStableContext.sol#L27>

Internal Pre-conditions

1. wUSDE WStable contract is deployed with sUSDe as the ERC4626 saving account (as planned in test deployment scripts)
2. Users hold wUSDE tokens and want to redeem them for USDe

External Pre-conditions

1. sUSDe contract has `cooldownDuration` != 0

Attack Path

This is not an attack but a systemic failure affecting all legitimate users:

1. User calls `redeem(amount, receiver, owner)` on wUSDE contract expecting 1:1 USDe withdrawal
2. `redeem()` calls `burn(amount, receiver, false)`
3. `burn()` attempts `savingAccount.withdraw(amount, receiver, address(this))`
4. sUSDe's `withdraw()` function reverts due to `ensureCooldownOff` modifier
5. User's transaction fails, preventing any withdrawal

Alternative path with degraded user experience:

1. User discovers they must call `burn(amount, receiver, true)` instead
2. This gives them sUSDe tokens, not the expected USDe tokens
3. User must then separately unstake sUSDe tokens
4. User must wait 7+ days for cooldown period to complete
5. Finally user can claim their original USDe amount

Impact

User Impact:

- **Complete withdrawal failure** because the primary `redeem()` function becomes unusable for all users
- **Fund lock-up** since users must wait 7+ days to access their deposited funds
- **Broken 1:1 invariant** because users cannot withdraw their deposits as USDe tokens instantly as promised
- **Poor user experience** because users must understand complex ERC4626 mechanics and perform multiple transactions

Protocol Impact:

- This Breaks the explicitly stated invariant **"Users are always able to withdraw 1:1 their deposits"**
- **Sherlock guideline violation:** Funds being locked for 7+ days
- **Design failure:** The WStable contract fails to fulfill its core purpose of providing instant stable token access

PoC

No response

Mitigation

Either replace sUSDe with an ERC4626 vault that supports instant withdrawals without cooldown periods or modify the WStable contract to handle cooldown periods in order to support similar vaults in the future

Issue M-12: Liquidation Fee is incorrectly computed

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/651>

Found by

OxFlare, Oxodus, Oxsh, OxxAristos, AMOW, Nyxx, ParthMandale, Pro_King, SaIntRobi, Tigerfrake, VictorUbi, auditorshambu, deadmanwalking, gh0xt, silver_eth, sll

Summary

The liquidation fee is incorrectly calculated as a percentage of the repaid debt (USGToRepay) rather than the liquidation bonus or profit (collateral value minus repaid debt), leading to scenarios where liquidators incur losses instead of profits, disincentivizing liquidations.

Finding Description

In the `_liquidate` function, <https://github.com/sherlock-audit/2025-08-usg-tangent/blob/main/tangent-contracts/src/USG/Market/abstract/MarketCore.sol#L441>

```
function _liquidate(LiquidateInput memory liquidateInput, ZapStruct calldata
↳ liquidateCall) internal returns (uint256, uint256, uint256, uint256) {
    uint256 collatAmountToLiquidate = liquidateInput.collatToLiquidate;

    _verifyCollatInputNotZero(liquidateInput.collatToLiquidate);

    uint256 debtSharesToRemove;
    uint256 USGToRepay;

    // Liquidate all
    if (collatAmountToLiquidate >= liquidateInput._collateralBalance) {
        collatAmountToLiquidate = liquidateInput._collateralBalance;
        USGToRepay = liquidateInput.userDebt;
        debtSharesToRemove = liquidateInput._userDebtShares;
    }
    // Liquidate partial
    else {
        USGToRepay = (collatAmountToLiquidate * liquidateInput.userDebt) /
↳ liquidateInput._collateralBalance;
        debtSharesToRemove = _convertToShares(USGToRepay,
↳ liquidateInput.newDebtIndex);

        // Ensure that the remaining debt is bigger than a minimum in order to
↳ leave profitable liquidation
        _verifyMinimumDebt(liquidateInput.userDebt - USGToRepay);
```

```

    }

    uint256 newUserDebtShares = liquidateInput._userDebtShares -
    ↪ debtSharesToRemove;
    // Modify the collateral balance, the user debt and the total debt
    _updateCollatAndDebts(
        liquidateInput.account,
        liquidateInput._collateralBalance - collatAmountToLiquidate,
        liquidateInput._totalCollateral - collatAmountToLiquidate,
        newUserDebtShares,
        liquidateInput._totalDebtShares - debtSharesToRemove
    );

    uint256 fee = _mulDiv(USGToRepay, liquidationFee, DENOMINATOR); //<<@audit

    _postLiquidate(collatAmountToLiquidate, USGToRepay + fee,
    ↪ liquidateInput.minUSGOut, liquidateCall);

    if (fee != 0) {
        _mintUSG(usr, controlTower.feeTreasury(), fee);
    }

    return (collatAmountToLiquidate, USGToRepay, fee, newUserDebtShares);
}

```

the liquidation fee is computed as:

```
fee = _mulDiv(USGToRepay, liquidationFee, DENOMINATOR);
```

where USGToRepay is the portion of the user's debt being repaid (proportional to the collateral liquidated). This fee is then added to the repayment amount in `_postLiquidate` (requiring the liquidator to effectively pay `USGToRepay + fee` while receiving collateral of potentially lower value) and minted to the treasury via `_mintUSG`. In a full liquidation scenario with a price drop, this results in the liquidator overpaying relative to the collateral's value. **For example: Scenario Parameters:**

- Initial collateral: \$5,000
- Initial debt: \$4,500
- Liquidation threshold: 95%
- Liquidation fee: 14%
- Collateral drops to: \$4,700
- Health factor: $0.992 < 1$ (liquidatable)

Current (Incorrect) Calculation:

- Liquidator receives: \$4,700 in collateral

- Liquidator pays: \$4,500 (debt) + \$630 (14% of \$4,500) = \$5,130
- Net loss to liquidator: \$5,130 - \$4,700 = \$430

Correct Calculation:

- Liquidation reward: \$4,700 - \$4,500 = \$200
- Fee should be: 14% × \$200 = \$28
- Liquidator pays: \$4,500 (debt) + \$28 (fee) = \$4,528
- Net profit to liquidator: \$172

Impact

- Liquidation Mechanism Failure: No rational liquidator will participate when guaranteed to lose money, effectively disabling the liquidation system.
- Protocol Insolvency Risk: Without functioning liquidations, undercollateralized positions accumulate bad debt, threatening protocol solvency.
- Cascading Liquidation Failures: During market stress, when liquidations are most needed, the broken incentive structure prevents liquidators from acting.

Mitigation

```
// Current (incorrect):
- uint256 fee = _mulDiv(USGToRepay, liquidationFee, DENOMINATOR);

// Should be (correct):
+ uint256 liquidationReward = collatAmountToLiquidate * _collateralPrice() / 1e18 -
  ↳ USGToRepay;
+ uint256 fee = _mulDiv(liquidationReward, liquidationFee, DENOMINATOR);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/117>

Issue M-13: Delayed Reward Cut Parameter Updates (Two-Cycle Enforcement Lag)

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/697>

Found by

Cybrid, ck, molaratai

Summary

The `RewardAccumulator::updateRCParams()` function exhibits unexpected behavior where new reward cut parameters only take effect after two full processing cycles instead of the expected one cycle. This occurs due to improper ordering of operations within the update function, creating a significant delay between administrative actions and their enforcement.

Root Cause

`updateRCParams()` calls `processRewards()` before updating `rcParams[market]`

```
function updateRCParams(address market, RParams calldata _rcParam) external
↪ onlyOwner {
    _verifyRCParams(_rcParam);
    processRewards(market, controlTower.feeTreasury()); // Uses OLD rcParams
    rcParams[market] = _rcParam;                       // Updates AFTER processing
}
```

```
function processRewards(address market, address harvestFeeReceiver) public {
    require(controlTower.isMarket(market), NotAMarketRewards());
    // Checkpoint the rewards
    _updateReward(market, address(0), 0, ICollateral(market).totalCollateral());

    // Claim the rewards linked to the collateral from the underlying protocol
    TokenAmount[] memory rewardAmounts = IMarketExternalActions(market).claimUn
↪   derlyingRewards(rewardTokens[market]);
    uint256 rewardTokensLength = rewardAmounts.length;
    @>   uint256 rewardCutPercentage = lastRewardCuts[market]; //process reward
↪   with old params

    @>   RParams memory _rcParams = rcParams[market]; //caches the old param

    // Computes and actualize the new reward cut
    @>   lastRewardCuts[market] = _calculateRC(USGOOracle.price_w(), _rcParams);
↪   //updates the next cycle call with old params
```

```

uint16 harvestFeePercentage = _rcParams.harvestFeePercentage;

// Stream rewards for every tokens
for (uint256 tokenIndex; tokenIndex < rewardTokensLength; ) {
    IERC20 rewardToken = rewardAmounts[tokenIndex].token;
    // Update streaming data for the current rewardToken
    (uint256 rewardCutAmount, uint256 harvesterAmount) =
        ↪ _processRewards(market, rewardToken,
        ↪ rewardAmounts[tokenIndex].amount, harvestFeePercentage,
        ↪ rewardCutPercentage);
    ...
}

```

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

1. Cycle 1: Admin calls `updateRCParams()` expecting new parameters to be staged for next cycle
 - `processRewards()` executes with old `rcParams`
 - `lastRewardCuts[market]` is calculated using old parameters
 - New `rcParams` are stored but won't be used yet
2. Cycle 2: Next `processRewards()` call
 - Still uses `lastRewardCuts` calculated with old parameters
 - Calculates new `lastRewardCuts` with new parameters (finally)
3. Cycle 3: Third `processRewards()` call
 - First time new parameters actually affect reward distribution

Impact

Parameter updates (e.g., during token depegging events) are delayed by two full cycles

PoC

No response

Mitigation

Reorder operations in `updateRCParams()` to update parameters before processing rewards:

```
function updateRCParams(address market, RCParams calldata _rcParam) external
↳ onlyOwner {
    // Verify if the new reward cut params are compliant
    _verifyRCParams(_rcParam);

+   // Update RC params BEFORE processing rewards
+   rcParams[market] = _rcParam;

    // Process the rewards (now with updated rcParams for next cycle calculation)
    processRewards(market, controlTower.feeTreasury());

-   // Update the new RC Params
-   rcParams[market] = _rcParam;
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/118>

Issue M-14: Edge-case USG prices will force reverts for functions relying on IRCalculator

Source: <https://github.com/sherlock-audit/2025-08-usg-tangent-judging/issues/727>

Found by

0xDelvine, 0xShoonya, coin2own

Summary

The integer division in computing `quotientFixedPoint` inside `_computeIR()` will cause it to become zero for edge-case USG prices as `_pow` will call `log_2(0)` and revert.

Root Cause

In `IRCalculator.sol:249` the calculation of `quotientFixedPoint` performs integer division before converting into 64.64 fixed-point, which truncates fractional values below 1 to 0. This 0 then propagates into `_pow()` `IRCalculator.sol:295`, where `log_2(0)` `IRCalculator.sol:296` is called, causing a revert due to the `require(x > 0)` check in `ABDKMath64x64.log_2`.

Internal Pre-conditions

1. The contract fetches a `usgPrice` value.
2. `USGPrice` needs to be less than `pMax * E12`.
3. The difference (`pMax * E12 - USGPrice`) must be smaller than (`pMax - pMin`)

External Pre-conditions

The oracle needs to return a `USGPrice` value extremely close to, but not more than `pMax * E12`

Attack Path

1. A user calls a function that calls `_computeIR()` to trigger the interest rate calculation.
2. The protocol fetches `USGPrice` from the oracle to use as an input in `_computeIR()`.
3. Inside `_computeIR()`, the calculation of `quotientFixedPoint` begins:

```
int128 quotientFixedPoint = ABDKMath64x64.divu(
    ((uint256(irParam.pMax) * E12) - USGPrice) / (irParam.pMax - irParam.pMin),
    E12
);
```

- Here, the **inner division** $((pMax * E12) - USGPrice) / (pMax - pMin)$ truncates to 0 when USGPrice is very close to $(pMax * E12)$.
 - This 0 is then passed as the numerator into `ABDKMath64x64.divu(0, E12)`.
4. Inside `divu`, the library tries to left-shift the numerator ($0 \ll 64$) to scale it into 64.64 fixed-point format. Since the numerator is already 0, the result of the shift remains 0. No meaningful fixed-point value can be produced.
 5. As a result, `quotientFixedPoint = 0`.
 6. `_computeIR()` passes `quotientFixedPoint = 0` into `_pow()`.
 7. `_pow()` calls `ABDKMath64x64.log_2(0)`, which reverts due to the `require(x > 0)` check inside `log_2`.

Impact

- The protocol cannot compute the interest rate whenever `usgPrice` approaches $(pMax * 1e12)$.
- This causes `_computeIR()` to revert, which blocks dependent functions from executing.
- As a result, users are unable to perform actions that require interest rate calculation i.e `withdraw`, `borrow`, `repay`, `liquidate`, `migrateFrom`, `migrateTo`.

PoC

Proof of Concept

1. Paste the following solidity test into `tangent-contracts/test/unit/IRCalculator`

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import {ABDKMath64x64} from "../../src/libs/ABDKMath64x64.sol";

contract IRBugTest is Test {
    uint256 constant E12 = 1e12;
    uint256 constant E13 = 1e13;
    uint256 constant ONE_ETHER = 1e18;
```

```

uint256 usgPrice = 999_999_999_999_980_001; // the edge case

struct IRParams {
    uint32 pMin;
    uint32 pMax;
    uint32 pInf;
    uint32 rMin;
    uint32 rMax;
    uint32 a1;
    uint32 a2;
    uint32 k;
    bool isHEC;
}

function _computeIR(uint256 USGPrice, IRParams memory irParam) public pure
↳ returns (uint216) {
    if (USGPrice <= uint256(irParam.pMin) * E12) {
        return uint216(irParam.rMax * E13);
    }
    if (USGPrice >= uint256(irParam.pMax) * E12) {
        if (irParam.isHEC) return 0;
        return uint216(irParam.rMin * E13);
    }

    int128 sigmaX = -ABDKMath64x64.mul(
        ABDKMath64x64.fromUInt(irParam.k),
        ABDKMath64x64.divi(
            int256(USGPrice) - int256(int32(irParam.pInf)) * int256(E12),
            int256(ONE_ETHER)
        )
    );
    int128 one = ABDKMath64x64.fromUInt(1);
    int128 sigma = ABDKMath64x64.div(one, one + ABDKMath64x64.exp(sigmaX));
    int128 alpha1 = ABDKMath64x64.divu(irParam.a1, 1000);
    int128 alpha = ABDKMath64x64.add(
        alpha1,
        ABDKMath64x64.mul(
            ABDKMath64x64.sub(ABDKMath64x64.divu(irParam.a2, 1000), alpha1),
            sigma
        )
    );

    int128 quotientFixedPoint = ABDKMath64x64.divu(
        ((uint256(irParam.pMax) * E12) - USGPrice) / (irParam.pMax -
↳ irParam.pMin),
        E12
    );

    uint256 irIncrement = ABDKMath64x64.mulu(

```

```

        ABDKMath64x64.mul(
            ABDKMath64x64.fromUInt(irParam.rMax - irParam.rMin),
            _pow(quotientFixedPoint, alpha)
        ),
        E13
    );

    return uint216(uint256(irParam.rMin) * E13 + irIncrement);
}

function _pow(int128 a, int128 b) private pure returns (int128) {
    return ABDKMath64x64.exp_2(ABDKMath64x64.mul(ABDKMath64x64.log_2(a), b));
}

// external wrappers so we can catch reverts in try/catch
function callLog2(int128 x) external pure {
    ABDKMath64x64.log_2(x);
}

function callCompute(uint256 price, IRParams memory params) external pure {
    _computeIR(price, params);
}

function test_RevertWhen_QuotientIsZero() public {
    IRParams memory params = IRParams({
        pMin: 980_000,
        pMax: 1_000_000,
        pInf: 990_000,
        rMin: 1,
        rMax: 5,
        a1: 500,
        a2: 1500,
        k: 1,
        isHEC: false
    });

    // Prove integer division collapse
    uint256 rawNumerator = (uint256(params.pMax) * E12) - usgPrice;
    uint256 rawDenom = params.pMax - params.pMin;
    uint256 afterIntDiv = rawNumerator / rawDenom;
    int128 quotientFixedPoint = ABDKMath64x64.divu(afterIntDiv, E12);

    console.log("rawNumerator:", rawNumerator);
    console.log("rawDenom:", rawDenom);
    console.log("afterIntDiv:", afterIntDiv);
    console.log("quotientFixedPoint:", int256(quotientFixedPoint));

    assertEq(afterIntDiv, 0, "Expected afterIntDiv == 0");
    assertEq(quotientFixedPoint, 0, "Expected quotientFixedPoint == 0");
}

```

```

    // Catch revert in log_2
    try this.callLog2(quotientFixedPoint) {
        fail("Expected log_2(0) to revert, but it did not");
    } catch (bytes memory revertData) {
        console.log("Caught revert in log_2, data length:", revertData.length);
        assertEq(revertData.length, 0, "Revert data should be empty (require
            ↪ log_2 check : x > 0 failed)");
    }

    // Catch revert in _computeIR
    try this.callCompute(usgPrice, params) {
        fail("Expected _computeIR to revert, but it did not");
    } catch (bytes memory revertData) {
        console.log("Caught revert in _computeIR, data length:",
            ↪ revertData.length);
        assertEq(revertData.length, 0, "Revert data should be empty (require
            ↪ log_2 check: x > 0 failed)");
    }
}
}
}

```

2. Run the following command:

```
forge test --match-path test/unit/IRCalculator/bugProof.t.sol -vvv
```

```

forge test --match-path test/unit/IRCalculator/bugProof.t.sol -vvv
[] Compiling...
No files changed, compilation skipped
forge test --match-path test/unit/IRCalculator/bugProof.t.sol -vvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/unit/IRCalculator/bugProof.t.sol:IRBugTest
[PASS] test_RevertWhen_QuotientIsZero() (gas: 26384)
Logs:
  rawNumerator: 19999
  rawDenom: 20000
  afterIntDiv: 0
  quotientFixedPoint: 0
  Caught revert in log_2, data length: 0
  Caught revert in _computeIR, data length: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.33ms (646.97µs CPU
  ↪ time)

Ran 1 test suite in 51.24ms (1.33ms CPU time): 1 tests passed, 0 failed, 0 skipped
  ↪ (1 total tests)

```

Mitigation

Add an explicit check before performing the division in `_computeIR` to ensure that the numerator is **greater than or equal to** the denominator. If the numerator is smaller, the function should handle this case gracefully to avoid unintended reverts.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
<https://github.com/Tangent-labs/tangent-contracts/pull/119>

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.