Pashov Audit Group

Tangent Security Review

October 30th 2025 - November 12th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Tangent

Tangent is a collateralized lending market that lets users deposit assets, borrow USG, leverage positions, repay, withdraw, migrate positions, and perform complex liquidations or zaps through external routing contracts. It coordinates collateral accounting, debt-share mechanics, pricing oracles, liquidation thresholds, and reward distribution while exposing a wide surface of advanced actions such as flash-mint-based leverage, multi-asset zaps, migrator-controlled state moves, and full liquidation flows.

5. Executive Summary

A time-boxed security review of the **Tangent-labs/tangent-contracts** repository was done by Pashov Audit Group, during which **Bretzel**, **t.aksoy**, **btk**, **pontifex**, **jesjupyter** engaged to review **Tangent**. A total of **23** issues were uncovered.

Protocol Summary	
Project Name	Tangent
Protocol Type	CDP stablecoin
Timeline	October 30th 2025 - November 12th 2025

Review commit hash:

- 687d78d70e48d68a3438c9148b015fa9d0904a8e (Tangent-labs/tangent-contracts)

Fixes review commit hash:

- fc8a8f822d67a7a99ab6e2f6764b643fbd4e1c70 (Tangent-labs/tangent-contracts)

Scope

`Collateral.sol` `DebtIR.sol` `MarketCore.solMarketExternalActions.sol` `PauseSettings.sol` `BasicERC20Market.sol` `ConvexCrvLPMarket.sol` `ConvexFxnLPMarket.sol` `OracleCryptoSwap.sol` `OracleDuoPoolStable.sol` `OracleBase.sol` `OraclePendlePT.sol` `OracleChainlinkWrapper.sol` `OracleCoinFromCurveLP.sol` `OracleERC4626.sol` `PendlePTRouter.sol` `TAN.sol` `USG.sol` `VsTAN.sol` `WStable.sol` `LightOwnable.sol` `LightReentrancyGuardTransient.sol` `ZappingUtil.sol` `ControlTower.sol` `IRCalculator.sol` `MarketCreator.sol` `Migratoor.sol` `RewardAccumulator.sol` `ZappingProxy.sol`

6. Findings

Findings count

Severity	Amount
Medium	3
Low	20
Total findings	**23**

Summary of findings

ID	Title	Severity	Status
[M-01]	Waste or unfairness from incorrect reward distribution	Medium	**Resolved**
[M-02]	Losses in extreme scenarios due to missing collateral check	Medium	**Resolved**
[M-03]	Failure of USDT token support	Medium	**Resolved**
[L-01]	WStable mint/burn misused ERC4626 preview functions	Low	**Resolved**
[L-02]	Off-by-one on unlock: `== endLockTime` cannot unlock	Low	**Resolved**
[L-03]	Kick incentive can round to zero (no motivation to kick small locks)	Low	**Resolved**
[L-04]	Double-floor rounding in borrow path slightly favors users	Low	**Resolved**
[L-05]	"No-fail" path may return invalid price if primary fails without fallback	Low	**Resolved**
[L-06]	Division-by-zero risk in `IRCalculator` when `irParam.pMax` equals `irParam.pMin`	Low	**Resolved**
[L-07]	Conflicting liquidation fee limits in `MarketCore` init and `Collateral`	Low	**Resolved**
[L-08]	Off-by-one cap prevents setting max allowed liquidation fee in `Collateral`	Low	**Resolved**
[L-09]	Unlimited minting in `claimRewards()` if `maxWithdraw()` returns max	Low	**Resolved**

ID	Title	Severity	Status
[L-10]	Critical setter functions lack event emissions for parameter changes	Low	**Resolved**
[L-11]	Fee calculation inconsistency in `updateRCParams()`	Low	Acknowledged
[L-12]	Loss of backing risk when underlying ERC4626 share price decreases	Low	Acknowledged
[L-13]	Slippage protection fails during partial liquidation	Low	**Resolved**
[L-14]	Zero repayment risk from rounding in partial liquidation	Low	**Resolved**
[L-15]	Risk in OracleERC4626 due to ignored withdraw fees	Low	Acknowledged
[L-16]	Incorrect Unit Validation for `startCutPrice`	Low	**Resolved**
[L-17]	Miscalculation of leftover rewards when `processRewards` is called with zero total supply	Low	**Resolved**
[L-18]	Cross-market socialization of bad debt via global, price-driven reward cut	Low	Acknowledged
[L-19]	Liquidations become unprofitable when USG trades at a premium	Low	Acknowledged
[L-20]	`ZappingProxy` cannot receive ETH refunds resulting in failed zaps	Low	**Resolved**

Medium findings

[M-01] Waste or unfairness from incorrect reward distribution

Severity

Impact: Medium

Likelihood: Medium

Description

Both `VsTAN.sol` and `RewardAccumulator.sol` implement Synthetix-style staking reward systems but have flaws when `totalSupply (or totalCollateral) is zero`.

- When rewards are added during a zero-supply period, when leftover rewards from the previous period are included in the new `rewardRate`, wasting rewards that were never distributed.

```
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;
}
```

When `totalSupply = 0`, no rewards are actually distributed (no one to distribute to). However, leftover is calculated as if rewards for these periods were being distributed, effectively wasting rewards that were never distributed.

- Additionally, when the first user stakes after a zero-supply period, `lastUpdateTime` is not updated during the zero-supply period, causing all accumulated rewards from that period to be incorrectly distributed to the first staker, creating an unfair advantage.

```
if (_totalSupplyVsTan != 0) {  // or totalCollateral != 0
    rewardData[token].rewardPerTokenStored = _rewardPerToken(token);
    rewardData[token].lastUpdateTime =
_lastTimeRewardApplicable(rewardData[token].periodFinish);
}
```

```
return rewardData[_rewardToken].rewardPerTokenStored +
    (((_lastTimeRewardApplicable(...) - rewardData[_rewardToken].lastUpdateTime) *
      rewardData[_rewardToken].rewardRate * 1e18) / totalSupplyVsTan);
```

1. When `totalSupply = 0`, lastUpdateTime is NOT updated.
2. When the first user stakes, totalSupply is still zero before the update.
3. `lastUpdateTime` still reflects the old timestamp (before the zero-supply period) and is not updated.

4. After the stake, `totalsupply` is updated.
5. In the next stake/other operation, the `_rewardPerToken()` calculates rewards from `lastUpdateTime` to now, including the entire zero-supply period, and distribute it to the first user.

Recommendations

Zero-supply periods should be handled consistently and correctly.

[M-02] Losses in extreme scenarios due to missing collateral check

Severity

Impact: Medium

Likelihood: Medium

Description

The `_liquidate()` function calculates `collatValue` using oracle price and requires liquidators to burn `USGToRepay + fee`, but it doesn't verify that the collateral value is sufficient to cover the debt repayment.

While `minUSGOut` provides slippage protection for the swap and liquidation fee, it doesn't protect against scenarios where the oracle-based `collatValue` is less than `USGToRepay`.

In extreme price movements or oracle staleness, liquidators may be forced to burn more USG than they receive from selling the collateral, resulting in losses. The protocol should add a `minCollatValue` check to ensure liquidators are protected, or reject liquidations when `collatValue < USGToRepay` and route them to `seizeCollateral()` instead.

```
if (collatValue > USGToRepay) {
    // Fee is taken on the liquidation profits
    uint256 delta = collatValue - USGToRepay;
    fee = (liquidationFee * delta) / DENOMINATOR;
}
```

We can observe that there is no Protection When `collatValue < USGToRepay`. This could cause the liquidator to bear the risk of price discrepancy in extreme price movement scenarios.

Recommendations

Add a minimum collateral value check to protect liquidators or require the `collatValue` to always be greater than the `USGToRepay`.

[M-03] Failure of USDT token support

Severity

Impact: Medium

Likelihood: Medium

Description

The PendlePTRouter contract will fail when interacting with tokens like USDT on mainnet.

This happens because the internal functions (_approveIfNotAllowed and _transferFrom) use raw ERC20 calls (token.approve(...) and token.transferFrom(...)). Tokens like USDT do not return a boolean value on success. When Solidity calls these functions the transaction reverts, treating the non-standard behavior as a failure.

Recommendations

Replace the raw calls in your internal helper functions with the SafeERC20 versions.

Low findings

[L-01] WStable mint/burn misused ERC4626 preview functions

The `WStable` contract misuses ERC4626's `previewMint()` and `previewWithdraw()` functions for conversions in `mint()` and `burn()` when interacting with the `_savingAccount`. These preview functions round up to protect the original vault (unfavorably for users), but at the WStable layer, this protection does not work as expected. Instead, it inadvertently favors WStable users by allowing up to 1 more wei of value across mint/burn cycles.

- `mint()` (`isSaving == true`): `_savingAccount.previewMint(amountIn)` uses an upward-rounded estimate of assets needed for the deposited shares. Users can receive up to 1 extra wei of WStable value than the shares' true worth.
- `burn()` (`isSaving == true`): `_savingAccount.previewWithdraw(amount)` uses an upward-rounded estimate of shares for the burned amount. Users can get up to 1 more wei of shares than required.

Didn't find impact as the shares will accrue underlying, so all users can exit with `burn()` (`isSaving == true`).

Replace with standard convert functions for accurate valuations.

[L-02] Off-by-one on unlock: `== endLockTime` cannot unlock

In `VsTAN.unlock`, the guard uses a strict `<`:

```
require(endLockTime < block.timestamp, LockNotOver());
```

This prevents the owner from unlocking **at the exact expiry** (`block.timestamp == endLockTime`), effectively forcing a one-block wait.

Switch to a non-strict check to allow unlock at expiry:

```solidity require(endLockTime <= block.timestamp, LockNotOver());

### [L-03] Kick incentive can round to zero (no motivation to kick small locks)

In `VsTAN.kickPosition`, the incentive is computed with integer division:

```
uint256 kickIncentivization = (_kick.percentage * amount) / 100_000;
```

For small `amount` or low `percentage`, this rounds to `0`, leaving the kicker unpaid and reducing liveness (positions linger past delay).
```

Enforce minimum position & incentive:

- Keep `minLockAmount` and `minKickIncentive` (wei TAN). Any new/modified lock must satisfy `amount ≥ minLockAmount`.
- On config updates, enforce `(kick.percentage * minLockAmount) / 100_000 ≥ minKickIncentive` (else revert).

[L-04] Double-floor rounding in borrow path slightly favors users

In the borrow flow, both conversions use floor rounding, which (a) slightly underestimates current debt and (b) mints slightly fewer debt shares for the same borrowed amount (≤1 wei USG per operation). While economically negligible, this bias is systematic.

```
    function _borrow(address receiver, uint256 USGToBorrow, uint256 collatAmount, bool
isLeverage) internal returns (uint256, uint256) {
        _verifyIsBorrowNotPaused();
        _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);
<..>

    function _convertToAmount(uint256 debtShares, uint256 index) internal pure returns
(uint256) {
        return _mulDiv(debtShares, index, RAY);
    }
<..>
    function _convertToShares(uint256 debt, uint256 index) internal pure returns (uint256) {
        return _mulDiv(debt, RAY, index);
    }
```

Keep `amount` with floor but mint debt shares with **ceil** (e.g., `shares = mulDivRoundingUp(amount, RAY, index)`), or use OZ's `Math.mulDiv(amount, RAY, index, Math.Rounding.Up)`, ensuring symmetric, protocol-friendly accounting.

[L-05] "No-fail" path may return invalid price if primary fails without fallback

In the `OracleChainlinkWrapper`'s "no-fail" branch without a fallback, the wrapper may return the raw (invalid) price, including **0**, which can distort liquidation logic.

Never return zero/invalid values. Enforce a positive price (and freshness) or use a cached last-good value; otherwise revert.

[L-06] Division-by-zero risk in `IRCalculator` when `irParam.pMax` equals `irParam.pMin`

The interest-rate configuration allows `irParam.pMax` to equal `irParam.pMin`. Any calculation that divides by the spread `(irParam.pMax - irParam.pMin)` will then revert due to division by zero:

```
            ((uint256(irParam.pMax) * E12) - USGPrice) / (irParam.pMax - irParam.pMin),
```

This can temporarily DoS operations that rely on the interest-rate calculator until governance updates parameters, since these parameters are mutable post-initialization.

```
    function _verifyIRParams(IRParams calldata _irParam) internal pure {
        require(_irParam.a1 <= 20_000, A1TooBig());
        require(_irParam.a2 <= 20_000, A2TooBig());
        require(_irParam.k <= 20_000, KTooBig());
        require(_irParam.rMax <= 400_000, RMaxTooBig());
        require(_irParam.rMin <= _irParam.rMax, RMinBiggerThanRMax());
        require(_irParam.pMin <= _irParam.pInf, PMinBiggerThanPInf());
        require(_irParam.pInf <= _irParam.pMax, PInfBiggerThanPMax());
        require(_irParam.pMax <= 1_000_000, PMaxBiggerThanOneDollar());
    }
```

Recommendation: Enforce a strict inequality at configuration time: require `irParam.pMax > irParam.pMin` in the initializer.

[L-07] Conflicting liquidation fee limits in `MarketCore` init and `Collateral`

`MarketCore.initialize()` validates `liquidationFee` against a permissive upper bound (~80%):

```
        require(_marketInit.liquidationFee < 80_000, LiquidationFeeTooHigh());
```

While the `Collateral` layer enforces a stricter cap of 15%:

```
        require(_liquidationFee < 15_000, LiquidationFeeTooHigh());
```

This inconsistency allows a market to be initialized with a `liquidationFee` value that passes `MarketCore.initialize()` but violates the intended protocol behavior set by `Collateral`.

Recommendation: Align `MarketCore.initialize()` validation with the stricter invariant by enforcing `liquidationFee <= 15_000`.

[L-08] Off-by-one cap prevents setting max allowed liquidation fee in `Collateral`

The `Collateral` base contract enforces an upper bound on `liquidationFee`, but the bound uses a strict less-than check that prevents setting the fee to the intended maximum (15%):

```
function setLiquidationFee(uint256 _liquidationFee) external onlyOwner {
    require(_liquidationFee < 15_000, LiquidationFeeTooHigh());
    liquidationFee = _liquidationFee;
}
```

As a result, governance cannot configure `liquidationFee` to exactly 15% and will encounter a revert when attempting to do so, despite this being within the documented limit.

Recommendation: - Change the validation to:

```
    require(_liquidationFee < 15_000, LiquidationFeeTooHigh());
```

[L-09] Unlimited minting in `claimRewards()` if `maxWithdraw()` returns max

The `claimRewards()` function in `WStable.sol` uses `maxWithdraw()` from the underlying ERC4626 vault to calculate rewards.

However, many ERC4626 implementations return `uint256.max` when there are no withdrawal limits, which causes `claimRewards()` to mint an unbounded amount of tokens to the fee treasury. When `maxWithdraw()` returns `uint256.max`, the calculation `totalStableStaked - dueAmount` results in a value close to uint256.max, causing massive token inflation and complete destruction of the token's economic model.

```
function mint(uint256 amountIn, address receiver, bool isSaving) public {
    require(amountIn != 0, ZeroAmount());

    uint256 amountToMint = amountIn;

    if (isSaving) {
        IERC4626 _savingAccount = savingAccount;
        amountToMint = _savingAccount.previewMint(amountIn);
        _savingAccount.transferFrom(msg.sender, address(this), amountIn);
    } else {
        stable.transferFrom(msg.sender, address(this), amountIn);
        savingAccount.deposit(amountIn, address(this));
    }

    // Mints the amount of WStable for the receiver
    _mint(receiver, amountToMint);
}
```

Thus, the function may have compatibility issues with a lot of ERC4626 vaults, and this should be noted.

[L-10] Critical setter functions lack event emissions for parameter changes

Multiple `setter/update` functions that modify critical protocol parameters do not emit events when values are changed. This makes it difficult to track parameter changes off-chain, monitor governance actions, and maintain an audit trail of protocol configuration updates.

Take the following examples:

```
function setCollatOracle(IPriceOracle _collatOracle) external onlyOwner {
    collatOracle = _collatOracle;
}
```

```
function setMaxLTV(uint256 _maxLTV) external onlyOwner {
    require(_maxLTV < liquidationThreshold, MaxLTVBiggerThanLiquidationThreshold());
    maxLTV = _maxLTV;
}
```

```
function setMaxMarketDebt(uint256 _maxMarketDebt) external onlyOwner {
    maxMarketDebt = _maxMarketDebt;
}
```

[L-11] Fee calculation inconsistency in `updateRCParams()`

When updateRCParams() is called, the function updates rcParams[market] before processing pending rewards. As a result, processRewards() uses:

-The new harvestFeePercentage, -The old rewardCutPercentage (from lastRewardCuts[market]).

```
        RCParams memory _rcParams = rcParams[market];

        // Computes and actulize the new reward cut
        lastRewardCuts[market] = _calculateRC(USGOracle.price_w(), _rcParams);

        uint16 harvestFeePercentage = _rcParams.harvestFeePercentage;
```

This leads to a temporary mismatch where harvesters' fees are charged according to the new parameters, while reward cuts are based on the old configuration.

Recommendations

Ensure both reward cuts and harvester fees are calculated for old parameter set.

[L-12] Loss of backing risk when underlying ERC4626 share price decreases

The WStable contract assumes that the underlying ERC4626 vault (savingAccount) only increases in value (yield-accruing). However, if the vault's share-to-asset ratio decreases due to a loss, depeg, or other risks, the accounting of WStable becomes inconsistent with the underlying assets.

During minting, the contract mints WStable 1:1 against the deposited stable tokens, regardless of the current ERC4626 exchange rate. But when the share price of the savingAccount drops, there may not be enough assets to cover all outstanding WStable tokens. Users who attempt to withdraw or redeem after the loss may not be able to withdraw their full amount.

Example:

> Initially: 100 shares = 110 assets → user deposits 110 assets and receives 110 WStable.
>
> Later: vault ratio drops to 100 shares = 100 assets → total supply = 110 WStable but only 100 assets are available.
>
> Users attempting to withdraw will revert

Implement explicit handling for negative yield or ensure the underlying vault share ratio doesn't decrease.

[L-13] Slippage protection fails during partial liquidation

The liquidation function uses maxUSGToBurn and minUSGOut parameters to protect the liquidator from excessive slippage. However, these protections can become ineffective if another partial liquidation occurs before the caller's transaction executes. Specifically, the function adjusts the liquidation amount as follows:

```
if (collatAmountToLiquidate >= liquidateInput._collateralBalance) {
    collatAmountToLiquidate = liquidateInput._collateralBalance;
    USGToRepay = liquidateInput.userDebt;
}
```

If the user's collateral has already been reduced by a previous liquidation, collatAmountToLiquidate is capped to a smaller amount, while the caller's maxUSGToBurn remains based on stale state (the pre-liquidation ratio).

As a result: - maxUSGToBurn may no longer accurately bound the actual repayment amount per unit of collateral, failing to protect the liquidator from overpaying.

- minUSGOut may fail since less collateral is liquidated than expected, leading to smaller output amounts.

Implement slippage protection relative to collateral units, not just total USG amounts.

[L-14] Zero repayment risk from rounding in partial liquidation

In the _liquidate() function, the following line performs integer division: `USGToRepay = (collatAmountToLiquidate * liquidateInput.userDebt) / liquidateInput._collateralBalance;` Because Solidity integer division truncates toward zero, very small collatAmountToLiquidate values can cause USGToRepay to evaluate to 0. This could allow a liquidation call that reduces collateral balances but does not actually repay any debt (or remove any debt shares). While such a scenario is generally unprofitable, if the collateral price is extremely high, even small collateral amounts could yield marginal gains, making this edge case worth guarding against.

Add a sanity check to ensure that the resulting debtSharesToRemove is nonzero before proceeding.

[L-15] Risk in OracleERC4626 due to ignored withdraw fees

The current implementation of OracleERC4626 uses the ERC4626 function convertToAssets(1e18) to determine the underlying asset amount per share. This approach does not account for withdrawal or redemption fees, potentially leading to an overestimation of the vault token’s true redeemable value. The convertToAssets() function provides a mathematical conversion between shares and assets, excluding any fees or slippage. Therefore, using it in a price oracle assumes a zero-fee vault, which can cause inflated price reports.

Replace the conversion call with previewRedeem() to accurately reflect redemption fees.

[L-16] Incorrect Unit Validation for `startCutPrice`

The RewardAccumulator contract validates startCutPrice and endCutPrice parameters against 1e18, implying that these values should be in 18-decimal precision. `require(_rcParam.startCutPrice <= 1e18, StartCutPriceTooHigh());` However, in the function _calculateRC, both values are multiplied by 1e12, meaning the implementation actually expects them to be stored as 6-decimal prices. This inconsistency means the upper-bound validation check (<= 1e18) is applied in the wrong scale, allowing overly large input values and breaking the intended reward cut curve.

[L-17] Miscalculation of leftover rewards when `processRewards` is called with zero total supply

In the processRewards function for both VsTAN and RewardAccumulator contracts, when it is called again while totalSupplyVsTan == 0 (i.e., no active positions), the contract sets a new rewardRate assuming that previous rewards were fully distributed. However, since _updateReward skips updating when the total supply is zero, no rewards are actually distributed.

`uint256 leftover = (rData.periodFinish - timestamp) * rData.rewardRate;` This assumes all prior rewards were streamed correctly until timestamp, which is inaccurate if there were no stakers. As a result, some tokens corresponding to the elapsed period become effectively locked in the contract and are never claimable.

Use the time since the last update, not the current timestamp, to calculate remaining undistributed rewards.

[L-18] Cross-market socialization of bad debt via global, price-driven reward cut

The reward cut applied by `RewardAccumulator` is a function of USG price only:

```
// RewardAccumulator.processRewards / processMultiRewards
uint256 USGPrice = USGOracle.price_w();
lastRewardCuts[market] = _calculateRC(USGPrice, rcParams[market]); // global, price-based
```

and `_calculateRC` itself depends solely on `USGPrice` thresholds/steps, not on any market's `badDebt` state. As a result, if one market accrues bad debt and drags USG price down, the increased reward cut is applied to all markets, effectively socializing the cost across unrelated markets. This contradicts the documentation statement that "interest and earnings might be used to repay the bad debt" of the specific market, suggesting targeted recovery rather than global dilution (https://docs.tangent.finance/docs/usg/health#bad-debt).

Recommendations

Decouple bad-debt recovery from the global price signal:

- Add a per-market component to the cut (e.g., `cut = max(cutByPrice(USG), cutByDebtRatio(market))`), where `cutByDebtRatio` rises with that market's `badDebt` / collateral value.
- After conversion in `processRewards`, earmark the market's portion and call a dedicated hook (e.g., `market.repayBadDebtFromRewards(amountUSG)`) until `badDebt == 0`. Keep other markets' rewards unaffected by a single market's loss.

[L-19] Liquidations become unprofitable when USG trades at a premium

The markets value USG-denominated debt at $1 per USG in health-factor / borrow-limit paths, while liquidators must source USG at the market price (P > 1). This compresses liquidation margins: (profit = gamma - (P - 1) - (fees && slippage)) and can turn liquidations uneconomic even for underwater positions (worse with leveraged loops).

```
    function _healthRatio(uint256 userDebt_, uint256 collateralBalance, uint256 collatPrice)
internal view returns (uint256) {
        if (userDebt_ != 0) {
>>          return (collateralBalance * 10 ** (18 - collatDecimals) * collatPrice *
liquidationThreshold) / (userDebt_ * DENOMINATOR);
        }
        return MAX_UINT; // Fully healthy if no debt
    }
```

Recommendations

Account for the effective USG price when valuing debt in HF/limit and liquidation math so that liquidation incentives remain positive during USG premiums e.g., use (P_eff = max(USG_TWAP, 1)).

[L-20] `ZappingProxy` cannot receive ETH refunds resulting in failed zaps

`ZappingProxy` acts as the routing shim that forwards user-supplied assets and calldata to arbitrary liquidity routers through `zapProxy()` @ZappingProxy.sol. When the zap path uses ETH as the input asset, `msg.value` is forwarded to the router, and the proxy later attempts to sweep any leftover ETH to `controlTower.feeTreasury()`.

```
        } else {
            balanceTokenInLeft = address(this).balance;
            if (0 != balanceTokenInLeft) {
                (bool isSuccess, ) = payable(controlTower.feeTreasury()).call{value:
balanceTokenInLeft}("");
                require(isSuccess, ZapCallError(""));
            }
        }
```

However, `ZappingProxy` provides neither a `receive()` nor `fallback()` function, so any ETH that a router tries to refund back to the proxy causes the router call to revert immediately. The entire zap therefore, fails even though the router sought only to return a change. This reproduces the previously reported “M-5 from sherlock,” demonstrating the known issue remains unresolved.

Recommendations

Implement a payable `receive()` function so router integrations can return leftover funds without reverting.