Pashov Audit Group

Tangent Security Review

December 8th 2025 - December 10th 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 **Ch_301**, **jesjupyter**, **t.aksoy**, **0xBugSlayer** engaged to review **Tangent**. A total of **10** issues were uncovered.

Protocol Summary	
Project Name	Tangent
Protocol Type	CDP Stablecoin
Timeline	December 8th 2025 - December 10th 2025

Review commit hash:
- 6fdbf1294386e8aa8bab9ec2483147be2bd61cee (Tangent-labs/tangent-contracts)

Fixes review commit hash:
- b79c0f1708aedcb0b17985faf28d52de97571b44 (Tangent-labs/tangent-contracts)

Scope

`CurveGaugeMarket.sol` `StakeDaoVaultV2Market.sol` `ConvexCrvLPMarket.sol` `ConvexFXNLPMarket.sol` `USG.sol` `ControlTower.sol` `IRCalculator.sol` `Migratoor.sol` `MarketCreator.sol` `RewardAccumulator.sol` `MarketExternalActions.sol` `MarketCore.sol` `PauseSettings.sol` `DebtIR.sol` `Collateral.sol`

6. Findings

Findings count

Severity	Amount
Medium	2
Low	8
Total findings	**10**

Summary of findings

ID	Title	Severity	Status
[M-01]	Receipt-token + Zap incorrectly handled in liquidation	Medium	**Resolved**
[M-02]	`CurveGaugeMarket` does not collect CRV allocation as reward	Medium	**Resolved**
[L-01]	Implicit 1:1 exchange rate assumption between tokens and `collateral`	Low	Acknowledged
[L-02]	Missing referrer in `StakeDao` vault deposit causing lost rewards	Low	**Resolved**
[L-03]	Missing events for access control role changes	Low	**Resolved**
[L-04]	Receipt token handling error in leverage	Low	**Resolved**
[L-05]	`MarketCreator` is not calling `Migrator::setIsMarket`	Low	Acknowledged
[L-06]	No check if pool `lpToken` matches market `collateralToken`	Low	**Resolved**
[L-07]	DoS in `claimUnderlyingRewards` due to strict transfer requirements on all reward tokens	Low	**Resolved**
[L-08]	Permanent DoS in `claimExtraRewards` from wrong array input type	Low	**Resolved**

Medium findings

[M-01] Receipt-token + Zap incorrectly handled in liquidation

Severity

Impact: Medium

Likelihood: Medium

Description

When a user requests liquidation with isReceiptOut = true and provides a zap route, the code transfers the vault receipt token (not LP token) to the zapper proxy. Instructs the zapper proxy to swap the LP token address (collatToken), but the zapper only holds the receipt token. Since approvals are not given for the receipt tokens swap will fail.

```
    function _postLiquidate(uint256 usgToBurn, PostLiquidate memory postLiquidate, ZapStruct
calldata liquidationCall) internal {
        ...

        _transferCollateralWithdraw(liquidationCall.router != address(0) ?
address(_zappingProxy) : msg.sender, postLiquidate.collatAmountToLiquidate,
postLiquidate.isReceiptOut);

        if (liquidationCall.router != address(0)) {
            _zappingProxy.zapProxy(collatToken, usg, postLiquidate.minUsgOut, msg.sender,
liquidationCall);
        }
```

Recommendations

When isReceiptOut = true, and a zap is requested, use the receipt token address as tokenIn for zap contract.

[M-02] `CurveGaugeMarket` does not collect CRV allocation as reward

Severity

Impact: Medium

Likelihood: Medium

Description

The `gauge::claim_rewards()` function only claims permissionless rewards (according to the `LiquidityGaugeV6` docs). The problem here is the fact, that except the permissionless rewards, the gauges are eligible to receive CRV allocation, and some of them will receive such. If CRV rewards are accumulated by the gauges, it will be impossible to claim them since the contract doesn't implement such functionality at the moment. This can be seen in the `CurveGaugeMarket::_claimRewards` function:

```
  function _claimRewards() internal override {
        // Claim the rewards from the gauge
//TAG: Gets the per rewards but leaves accumulated CRV rewards
        IGauge(receiptToken).claim_rewards();
    }
```

Recommendations

When the `CurveGaugeMarket::_claimRewards` function is called, call the `Minter:mint` function to harvest CVR rewards, as described in the provided gauge docs.

Low findings

[L-01] Implicit 1:1 exchange rate assumption between tokens and `collateral`

The market contract architecture implicitly assumes a 1:1 exchange rate between `receipt` tokens and `collateral` tokens. This assumption is not documented and could lead to incorrect accounting or value transfer if **a future market implementation uses receipt tokens with a non-1:1 exchange rate** (e.g., yield-bearing vault tokens that accumulate value over time).

The `_withdraw` function in MarketCore calculates withdrawal amounts in collateral units and passes this value directly to `_transferCollateralWithdraw` :

```
function _withdraw(uint256 amountToWithdraw, bool isReceiptOut) internal {
    // ... validation logic ...
    _transferCollateralWithdraw(msg.sender, amountToWithdraw, isReceiptOut);
}
```

In market implementations like `CurveGaugeMarket and StakeDaoVaultV2Market` , when `isReceipt = true` , the code transfers the same amount of receipt tokens without any exchange rate conversion:

```
// CurveGaugeMarket.sol:43-52
function _transferCollateralWithdraw(address to, uint256 collatToWithdraw, bool isReceipt)
internal override {
    if (isReceipt) {
        IGauge(receiptToken).transfer(to, collatToWithdraw);
    } else {
        IGauge(receiptToken).withdraw(collatToWithdraw);
        collatToken.transfer(to, collatToWithdraw);
    }
}
```

This only works correctly if receipt tokens and collateral tokens maintain a 1:1 exchange rate.

Currently, all implemented markets (Curve Gauge, StakeDao Vault) appear to maintain 1:1 ratios. However, this is an implicit assumption that is not enforced or documented.

[L-02] Missing referrer in `StakeDao` vault deposit causing lost rewards

The `_postDeposit` function in `StakeDaoVaultV2Market` deposits collateral into the `StakeDao` Vault without specifying a referrer parameter. If StakeDao's referrer mechanism provides rewards or benefits, the protocol may be missing out on potential referral rewards that could accrue to the protocol treasury.

The current implementation calls the two-parameter version of `deposit`, which does not include the referrer parameter:

```
function _postDeposit(IERC20 _collatToken, bool isReceiptIn) internal override {
    if (!isReceiptIn) {
        // Deposit the whole balance of LP into the StakeDao Vault
        IStakeDaoVaultV2(receiptToken).deposit(_collatToken.balanceOf(address(this)),
address(this));
    }
}
```

However, the `IStakeDaoVaultV2` interface provides a three-parameter version that accepts a referrer:

```
function deposit(uint256 assets, address receiver, address referrer) external returns (uint256);
```

According to StakeDao's documentation, the referrer parameter is used for tracking and may be emitted in Accountant events. If StakeDao's referrer program provides rewards or benefits to referrers, the protocol should set itself or a relevant address as the referrer to capture these potential rewards.

[L-03] Missing events for access control role changes

The USG contract lacks event emissions for access control role modifications. Four setter functions (`setIsMinter, setIsBurner, setIsIRProducer, and setIsPegKeeper`) update important permission mappings without emitting events, making it impossible to track historical role changes off-chain. This reduces auditability, transparency, and monitoring capabilities for critical permission changes in the protocol.

```
    /**
     * @notice Adds or removes an address from the minter role.
     * @dev Updates the `isMinter` mapping. Callable only by the contract owner.
     * @param minter    The address whose minter status will be updated.
     * @param _isMinter `true` to assign the minter role, `false` to revoke it.
     */
    function setIsMinter(address minter, bool _isMinter) external onlyOwner {
        isMinter[minter] = _isMinter;
    }

    /**
     * @notice Adds or removes an address from the burner role.
     * @dev Updates the `isBurner` mapping. Callable only by the contract owner.
     * @param burner    The address whose burner status will be updated.
     * @param _isBurner `true` to assign the burner role, `false` to revoke it.
     */
    function setIsBurner(address burner, bool _isBurner) external onlyOwner {
        isBurner[burner] = _isBurner;
    }

    /**
     * @notice Adds or removes an address from the ir producer role.
     * @dev Updates the `isIRProducer` mapping. Callable only by the contract owner.
     * @param irProducer    The address whose ir Producer status will be updated.
```

```
     * @param _isIRProducer `true` to assign the ir producer role, `false` to revoke it.
     */
    function setIsIRProducer(address irProducer, bool _isIRProducer) external onlyOwner {
        isIRProducer[irProducer] = _isIRProducer;
    }

    /**
     * @notice Adds or removes an address from the peg keeper role.
     * @dev Updates the `isPegKeeper` mapping. Callable only by the contract owner.
     * @param pegKeeper    The address whose peg keeper status will be updated.
     * @param _isPegKeeper `true` to assign the peg keeper role, `false` to revoke it.
     */
    function setIsPegKeeper(address pegKeeper, bool _isPegKeeper) external onlyOwner {
        isPegKeeper[pegKeeper] = _isPegKeeper;
    }
```

[L-04] Receipt token handling error in leverage

The leverage() logic incorrectly handles the case where isReceiptIn = true. When isReceiptIn = true, both receipt tokens (user-provided) and collateral tokens (obtained from zapping USG → collateral) end up inside the market contract, but _postDeposit will not deposit anything, leaving these tokens idle and unstaked.

```
        uint256 collatBought = _zappingProxy.zapProxy(_usg, collatToken,
leverageIn.minCollatAmountOut, address(this), dumpUSGCall);

        uint256 stakedAmount = leverageIn.collatToDeposit + collatBought;

        // Performs same modification as in depositAndBorrow
        uint256 newUserDebtShares = _depositAndBorrow(stakedAmount, leverageIn.usgToFlashMint,
_collatToken, true, leverageIn.isReceiptIn);
```

Modify leverage flow to stake collateral tokens obtained from zapping USG.

[L-05] `MarketCreator` is not calling `Migrator::setIsMarket`

`setIsMarket` is a newly implemented function for the `Migrator` contract, and as its natspec suggests:

> Restricted to the contract owner or addresses with the MarketCreator role.

As of right now, the function is callable only manually by the owner of the contract, meaning every market whitelist should be done by him. To fix this, allow the function to be called by the `MarketCreator` contract and add the call to the `MarketCreator::_commonInitialize` function.

[L-06] No check if pool `lpToken` matches market `collateralToken`

When intializing the `ConvexCrvLPMarket`, the reward token is extracted from the hardcoded pool address, but `lpToken` and `collateralToken` are never checked against each other, which is a necessary condition for the market functionality.

To fix this, check if the `lpToken` from the pool is the same as the used `collateraToken`.

[L-07] DoS in `claimUnderlyingRewards` due to strict transfer requirements on all reward tokens

The `RewardAccumulator` contract manages a list of reward tokens for each market (`rewardTokens[market]`). When `processRewards` is called, it triggers `claimUnderlyingRewards` on the market contract. This function, implemented in `MarketExternalActions` (inheriting from `MarketCore`), iterates through the entire list of reward tokens and attempts to transfer the balance of each token from the market contract to the `RewardAccumulator`.

The transfer logic is implemented in `MarketCore._claimUnderlyingRewards`:

```
// USG/Market/abstract/MarketCore.sol

function _claimUnderlyingRewards(IERC20[] memory _rewardTokens) internal returns (TokenAmount[]
memory) {
    uint256 rewardLen = _rewardTokens.length;
    // ...
    for (uint256 i; i < rewardLen; ) {
        IERC20 rewardToken = _rewardTokens[i];
        uint256 balance = rewardToken.balanceOf(address(this));
        if (balance != 0) {
            rewardAmounts[counter++] = TokenAmount({token: rewardToken, amount: balance});

            // CRITICAL: If this safeTransfer fails, the entire transaction reverts
            rewardToken.safeTransfer(address(_rewardAccumulator), balance);
        }
        unchecked { ++i; }
    }
    // ...
}
```

If any token in the `_rewardTokens` list fails to transfer (e.g., `safeTransfer` reverts), the entire function call reverts. Since `processRewards` relies on this call succeeding to distribute *any* rewards, a single problematic token effectively freezes the reward distribution mechanism for that market.

The permissionless nature of Curve Gauges (LiquidityGaugeV6), which allows the gauge manager to add up to 8 arbitrary reward tokens via `add_reward`. If a token that pauses transfers or blacklists the Tangent market contract, the reward system breaks. So, If one reward token becomes untransferable (paused, blacklisted, or buggy), no rewards can be processed or claimed for that market.

Recommendations

Modify `_claimUnderlyingRewards` in `MarketCore.sol` to handle transfer failures gracefully. Instead of reverting, catch the failure and skip that specific token, allowing the processing of other valid rewards to proceed.

[L-08] Permanent DoS in `claimExtraRewards` from wrong array input type

The `StakeDaoVaultV2Market` aims to collect rewards, just as any other market. The problem here is the wrong function arguments used for calling `StakeDaoVaultV2::claim()` function. As can be seen in the implementation provided right here, the `claim` function that we aim to call requires array of type `address` and `claimExtraRewards` function requires array of type `IERC20` as the array is never converted to type `address`:

```
@> function claimExtraRewards(IERC20[] calldata rewards) external nonReentrant {
        // Claim the extra rewards
        IStakeDaoVaultV2(receiptToken).claim(rewards, address(this));
    }
```

This will lead to the impossibility of collecting rewards from `StakeDaoVaultV2`, leaving only the `Accountant` rewards claimable.

Recommendations

Either convert the input array into type `address` or require the input array to be type `address`. Also, correct the function inputs in the `IStakeDaoVaultV2` interface.