Bauchibred

Panoptic

Security Review Report
January, 2025

Table of Contents

Introduction

At the request of the Panoptic team, Bauchibred conducted a time-boxed security review of the protocol, focusing on the security aspects of its smart contract implementation. The review aimed to identify potential security vulnerabilities and bugs within the defined scope, while also providing additional recommendations where appropriate.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks, this security review, while thorough, cannot guarantee the complete absence of vulnerabilities as it is bound by time, resources, and expertise, and Bauchibred assumes no responsibility for any misbehaviour, bugs, or exploits affecting the audited code or deployment phase.

Furthermore, it should be understood that any changes to the audited code, including fixes of highlighted issues in this report, may introduce new vulnerabilities and require further auditing.

Executive Summary

About Bauchibred

Bauchibred is an independent security researcher who specializes in smart contract audits. Having found numerous security vulnerabilities and winning multiple audit contests, he is currently one of the top security researchers on Code4rena and other competitive platforms. Check his previous work here.

For security consulting reach out to him on X (previously twitter) - *@bauchibred*.

About Panoptic

Panoptic is a decentralized and permissionless options trading protocol built on Uniswap V3 and V4. They've taken a new and innovative approach that allows them to adapt a novel form of perpetual options into a DeFi protocol with oracle-free settlement. Instead of relying on thin and centralized order books, Panoptic takes the form of an advanced lending market for Uniswap positions.

Scope

- **Strictly the new changes** to the files below introduced from two pull requests (PR1 & PR2):
  - contracts/CollateralTracker.sol
  - contracts/PanopticPool.sol
  - contracts/libraries/Errors.sol
  - contracts/libraries/PanopticMath.sol
  - contracts/types/PositionBalance.sol
  - contracts/types/TokenId.sol
- And a fresh utility contract to aid migrations:
  - src/UniswapMigrator.sol

Repository details

For the diffs

Repository URL: https://github.com/panoptic-labs/panoptic-v1-core/

- **Pull requests**:
  - **PR1**: https://github.com/panoptic-labs/panoptic-v1-core/pull/25
  - **PR2** https://github.com/panoptic-labs/panoptic-v1-core/pull/26
- **Language**: Solidity

`UniswapMigrator.sol`

Repository URL: https://github.com/panoptic-labs/panoptic-v1-helper/

- **Commit hash**: 065ceab30f9646fd3cea07559f728913dda6b4ba
- **Mitigation review hash**: cb1d1e6dbefbda2e9f7a33bbe5a88e41301c63f7
- **Language**: Solidity

Methodology

Overall, the audit methodology involved a thorough manual code review focusing on security patterns, potential vulnerabilities, and edge cases, while also examining external dependencies and their interactions with the protocol.

Qualitative Analysis

Metric	Rating	Comments
Code complexity	**Excellent**	Project has kept code as simple as possible, reducing attack risks
Documentation	**Excellent**	Project is very well documented
Best practices	**Good**	Project generally follows best practices
Centralization risks	**Good**	If system parameters are chosen safely, centralization is mostly minimized

Issues Found

Severity	Count
Low	5
Info	4

Total: 9 issues found

Findings

Summary

ID	Description	Severity
L-1	New implementation of using last observed tick could cause brickage in normal conditions	Low
L-2	Enforced buffer DOS's users from being able to use their collateral	Low
L-3	Migrations do not have a valid deadline protection	Low
L-4	Enforcing BP_DECREASE_BUFFER when only using long leg premia causes reverts	Low
L-5	V3 migrations are not allowed to vaults with ETH as token0	Low
I-1	Remove redundant query of token's balance during migrations	Info
I-2	Array length mismatch would break migrations	Info
I-3	Whitelist swapAddresses that can be called during batch swaps	Info
I-4	Incorrect Natspec/documentation	Info

Low Severity Findings

L-1 New implementation of using last observed tick for Force exercising & settleLongPremium could cause brickage in fairly normal conditions

Description

First note the following:

- New implementation of force exercising now uses the data at `slot0` from the oracle contract, instead of `twaptick` as was previously done.
- Also another addition is the integration of a ~ 2% check to ensure that the oracle is not stale.

However the new implementation subtly leads to two different issues covered in this section.

In `PanopticPool.sol#forceExercise()` we use the `lastObservedTick` which is imminently gotten from `slot0` via `computeMedianObservedPrice`:

```
    function forceExercise(..snip) external {
        _validatePositionList(msg.sender, positionIdListExercisor, 0);
    //@audit-issue previously we used twaptick for force exercising the
accounts now we use the lastobservedtick instead.
        int24 lastObservedTick;
        {
            int24 fastOracleTick;
            int24 slowOracleTick;
            (fastOracleTick, slowOracleTick, lastObservedTick, ) =
PanopticMath.getOracleTicks( oracleContract(), s_miniMedian);
            if (
                Math.abs(lastObservedTick - fastOracleTick) >
MAX_TICK_DELTA_SUBSTITUTION ||
                Math.abs(lastObservedTick - slowOracleTick) >
MAX_TICK_DELTA_SUBSTITUTION
            ) revert Errors.StaleOracle();
        }

    function computeMedianObservedPrice(..snip) {
        // ..snip
    // the `ticks` array descends from the most recent Uniswap observation
prior to the sort
            int24 latestObservation = int24(ticks[0]);
            // get the median of the `ticks` array (assuming `cardinality`
is odd)
            return (int24(Math.sort(ticks)[cardinality / 2]),
```

```
latestObservation);
    }
```

First issue

We are trading off the `twapTick` and instead getting the data from the most recent Uniswap observation, which is also the most volatile as it can be considered the spot price.

This then means that *subtly*, force exercising depending on if the price weared off up/down, would then be unfair for the force exercisor/exercisee.

Second issue

This builds on the first, going to the new implementation of the Panoptic Pool, we see how we now enforce an atmost 2% check to ensure that the oracle is not stale during `forceExercise/settleLongPremium`:

```
    /// @notice The maximum allowed delta (~2%) between the lastObservedTick
and the slowOracleTick/fastOracleTick during
forceExercise/settleLongPremium.
    /// @dev Ensures token substitution between two accounts is settled
safely at a price close to market.
    int256 internal constant MAX_TICK_DELTA_SUBSTITUTION = 203;
```

Would be key to note that this bracket is actually 40% of the bracket left on liquidations.

Coupling with the fact that the force exercising uses the spot price, then we easily encounter reverts in trying to force exercise in fairly normal market conditions, considering the 2% buffer can easily be hit more often against the slow tick since the cardinality for this is larger.

Impact

As already hinted above, this can cause *unfair* exercising, or in the second case DOS at valid attempts of force exercising a distressed account, cause if we assume market having quite a natural fall, 2.5% over the cardinality of the slow oracle and 1.25% over the cardinality of the fast oracle, whereas this should be acceptable, we hit a revert here.

Recommended mitigation

For the first case, I assume this is intended implementation but still had to flag, for the second case however, since the intention of applying `MAX_TICK_DELTA_SUBSTITUTION` is to ensure that the token substitution is settled safely at a price close to market, then we can't have this be the same value for both the fast oracle and slow oracle, since arguably the fast oracle will be closer to the spot price than the slow oracle.

Team response

Acknowledged, we expect that force exercises will frequently revert for this reason. Force exercising is not a time-sensitive action, however, so this is acceptable (and arguably this is not worse than the current behavior given that the TWAP tick can be stale -- in both cases the price will eventually converge and allow a force exercise if the option stays out-of-range.

Mitigation Review

Acknowledged.

L-1 Enforced buffer DOS's users from being able to use their collateral

Description

When minting we enforce the 133% buffer however when liquidating we don't enforce any buffer so this means that whereas the users should be able to mint as much as their collaterals would allow them in as much as they are still in a *safe bracket (nonliquidatable)* to them, we deny them this ability and then have them over the 133% both which stops them from mining.

```
    /// @notice Multiplier in basis points for the collateral requirement in the event of a buying power decrease, such as minting or force exercising another user.
    uint256 internal constant BP_DECREASE_BUFFER = 13_333;
```

When minting options we check this:

```
        _checkSolvency(
            msg.sender,
            positionIdList,
            tickData,
            BP_DECREASE_BUFFER,
            usePremiaAsCollateral
        );
```

But when liquidating we don't check any buffer:

```
            _checkSolvencyAtTicks(
                liquidatee,
                positionIdList,
                currentTick,
                atTicks,
                NO_BUFFER,
                ASSERT_INSOLVENCY,
                COMPUTE_PREMIA_AS_COLLATERAL
            );
```

Which then means that users are being locked off from 33% of their collateral.

Recommended mitigation

Consider allowing users pass in their specified buffer in as much as it doesn't make them immediately liquidatable, i.e check it against the `NO_BUFFER` value

Team response

Acknowledged, this is intended behavior -- arguably it is somewhat of an overbearing restriction (which also exists on our live mainnet contracts), but we wanted to ensure users can only be liquidated by large price moves once they have minted their position

Mitigation Review

Acknowledged.

L-2 Migrations do not have a valid deadline protection

Description

When migrating V3 we pass in an infinite deadline, allowing for the transaction to linger in the mempool infinitely, the same is also the case when we are modifying the liquidity in V4 migrations.

In `migrateV3()` we decrease the liquidity while passing in an infinite deadline

```
        NFPM.decreaseLiquidity(
            INonfungiblePositionManager.DecreaseLiquidityParams({
                tokenId: tokenId,
                liquidity: liquidity,
                amount0Min: amount0Min,
                amount1Min: amount1Min,
                deadline: type(uint32).max
            })
        );
```

Which means that the transaction can linger in the mempool indefinitely and be executed at an unfavorable time to the migrator, considering the NFPM expects this deadline and checks it:

`NFPM#decreaseLiquidity()`

```
    function decreaseLiquidity(DecreaseLiquidityParams calldata params)
        external
        payable
        override
        isAuthorizedForToken(params.tokenId)
|>      checkDeadline(params.deadline)
```

```
        returns (uint256 amount0, uint256 amount1)
    {
        ...
    }
```

```
    modifier checkDeadline(uint256 deadline) {
        require(_blockTimestamp() <= deadline, 'Transaction too old');
        _;
    }
```

Similarly in `migrateV4()` we pass in an incorrect deadline when modifying the liquidity on V4_PM:

```
        V4_PM.modifyLiquidities(
            abi.encode(
                abi.encodePacked(uint8(Actions.BURN_POSITION),
uint8(Actions.TAKE_PAIR)),
                params
            ),
|>            block.timestamp
        );
```

As hinted earlier on, migrations could be executed at a disadvantage to the migrator, which could cause loss for the migrator, *albeit in* `$USD` *terms, considering we have the* `amount0min` *and* `amount1min` *limitation*.

Recommended mitigation

Allow the migrator to pass in a valid deadline.

Team response

Acknowledged, users should replace the transaction if it takes too long.

Mitigation Review

Acknowledged, however users should know there is still a possibility where even if/when they replace the transaction, the transaction could've lingered for enough time in the mempool to cause them loss in `$USD` terms.

L-3 Enforcing `BP_DECREASE_BUFFER` when only using long leg premia during withdrawals would cause reverts in some cases

Description

The new scope introduces a `usePremiaAsCollateral` parameter to the `validateCollateralWithdrawable` function in `PanopticPool`.

Now the `validateCollateralWithdrawable` function is called when attempting to wihdraw collateral from the CollateralTracker contract:

```
        // reverts if account is not solvent/eligible to withdraw
        _panopticPool().validateCollateralWithdrawable(
            owner,
            positionIdList,
            usePremiaAsCollateral
        );
```

Issue however is that this new implemented bool now means users can decide to withdraw only their long leg premia or both:

```
    /// @param usePremiaAsCollateral Whether to compute accumulated premia
for all legs held by the user for collateral (true), or just owed premia for
long legs (false)
```

Now the logic around withdrawals is only to revert **if the account is not solvent with the given `positionIdList`**.

But validating the solvency does not take the whole account into consideration when `usePremiaAsCollateral = false`:

```
    function validateCollateralWithdrawable(
        address user,
        TokenId[] calldata positionIdList,
        bool usePremiaAsCollateral
    ) external view {
        _validateSolvency(user, positionIdList, BP_DECREASE_BUFFER,
usePremiaAsCollateral);
    }
```

This then means that when a user sets `usePremiaAsCollateral = false` we are only validating their long leg premia and not their short leg premia, so even if their short leg premia would have them be solvent after the withdrawal, we still revert in checkSolvencyAtTicks, coupled with the fact that the `BP_DECREASE_BUFFER` is applied making the window even more narrow for reverts to occur on valid withdrawal attempts per Panoptic's standards.

Recommended mitigation

Quite complex to implement a check, but we can consider the following options:

If account own substantial collateral on both type of legs, then remove or reduce the buffer when `usePremiaAsCollateral = false`:

```
function validateCollateralWithdrawable(
    address user,
    TokenId[] calldata positionIdList,
    bool usePremiaAsCollateral
) external view {
    uint256 buffer = usePremiaAsCollateral ? BP_DECREASE_BUFFER : NO_BUFFER;
    _validateSolvency(user, positionIdList, buffer, usePremiaAsCollateral);
}
```

Alternatively, enforce that solvency checks always consider all premia regardless of `usePremiaAsCollateral` to maintain consistent safety margins:

```
function validateCollateralWithdrawable(
    address user,
    TokenId[] calldata positionIdList,
    bool usePremiaAsCollateral
) external view {
    // Always use true for usePremiaAsCollateral in withdrawals
    _validateSolvency(user, positionIdList, BP_DECREASE_BUFFER, true);
}
```

Or change the documentation to explicitly indicate that withdrawals would not be allowed depending on the type of legs one owns collateral if `usePremiaAsCollateral` is false.

Alternatively, just use a similar implementation as the solvency check during force exercises:

```
            _validateSolvency(
                 msg.sender,
                 positionIdListExercisor,
                 BP_DECREASE_BUFFER,
|>                 usePremiaAsCollateral.leftSlot() > 0
             );
```

Team response

Acknowledged, the intent is for the interface to detect whether the user is solvent or not prior to passing that flag in -- if they indeed require their short premium to execute a withdrawal, we can opt to set that flag in exchange for an increased gas cost.

Mitigation Review

Acknowledged.

L-4 V3 migrations are not allowed to vaults with ETH as `token0`

Description

Whereas in V4 migrations we allow ETH deposits to vaults with ETH as if the token0 is the native token, in V3 migrations this check is completely not implemented, disallowing direct migrations from V3 to a collateral tracker vault with ETH as token0.

In `migrateV4()`, we allow for deposits to vaults with ETH as token0:

```
    function migrateV4{
        // ..snip

        // Tokens are sorted alphanumerically, so native ETH is always
token0
        if (token0 == address(0)) {
            if (address(this).balance > 0)
                ct0.deposit{value: address(this).balance}
(address(this).balance, msg.sender);
        }

                    // ..snip
    }
```

In migrateV3() however we don't attach this check to allow for direct migrations to vaults with ETH as token0.

Impact

Considering that ETH would be quite common in V4 pairs, this would then mean that direct migrations to these pairs on V4 would not work

Recommended mitigation

Allow migrations to ETH pairs to via `migrateV3()`.

Team response

There are no native ETH pairs in Uniswap V3, so if we kept the logic similar between migrateV3 and V4 the current implementation would be correct, you would go from USDC/WETH on V3 to USDC/WETH on V4 (with the WETH token). However there is a use case for USDC/WETH -> USDC/ETH given that ETH will likely be more common in V4 pairs. Thanks for flagging this! I will try to add support for this next week.

Support for ETH migration from V3 has been added here.

Mitigation Review

Recommended fix has been applied, new implementation is sound and allows for direct migration to V4 ETH pairs from V3, there is a new bug case here, as refunds of surplus ETH are not sent back to the migrator after this migration.

Informational Severity Findings

I-1 Remove redundant query of token's balance during migrations

Description

During migrations we cache the balance of `token1` as we do that of `token0` before the deposit in the vault, however in the former's case we then re-query during the attempt at depositing instead of using the already cached value.

```
    function migrateV3(
        uint256 tokenId,
        uint256 amount0Min,
        uint256 amount1Min,
        address[] memory swapAddresses,
        bytes[] memory swapCalls,
        CollateralTracker ct0,
        CollateralTracker ct1
    ) external {
        if (NFPM.ownerOf(tokenId) != msg.sender) revert
PeripheryErrors.UnauthorizedMigration();
//..snip

|>          amountCollected =
IERC20Partial(token1).balanceOf(address(this));
        if (amountCollected > 0) {
            IERC20Partial(token1).approve(
                address(ct1),
|>                  IERC20Partial(token1).balanceOf(address(this))
            );
            ct1.deposit(amountCollected, msg.sender);
        }
    }
```

Similarly in `migrateV4()` , there is a redundant check for `token0 != address(0)` when handling token deposits. This check is unnecessary because the code path is already inside an else block that only executes when `token0 != address(0)`.

```
        // Tokens are sorted alphanumerically, so native ETH is always
token0
        if (token0 == address(0)) {
            if (address(this).balance > 0)
                ct0.deposit{value: address(this).balance}
(address(this).balance, msg.sender);
```

```
        } else {//@audit we only enter this block if token0 != address(0)
            uint256 amount0Collected =
IERC20Partial(token0).balanceOf(address(this));
            if (amount0Collected > 0) {
                if (token0 != address(0))
                    IERC20Partial(ct0.asset()).approve(address(ct0),
amount0Collected);
                ct0.deposit(amount0Collected, msg.sender);
            }
        }
```

Recommended mitigation

```
    function migrateV3(
        uint256 tokenId,
        uint256 amount0Min,
        uint256 amount1Min,
        address[] memory swapAddresses,
        bytes[] memory swapCalls,
        CollateralTracker ct0,
        CollateralTracker ct1
    ) external {
        if (NFPM.ownerOf(tokenId) != msg.sender) revert
PeripheryErrors.UnauthorizedMigration();
// ..snip
        amountCollected = IERC20Partial(token1).balanceOf(address(this));
        if (amountCollected > 0) {
            IERC20Partial(token1).approve(
                address(ct1),
-                IERC20Partial(token1).balanceOf(address(this))
+                amountCollected
            );
            ct1.deposit(amountCollected, msg.sender);
        }
    }
```

Alternatively to make the function more efficient do not cache the balances and instead just pass them direct, saving gas, as is currently done redundantly.

And remove the redundant check since we already know `token0 != address(0)` in this code path:

```
    if (token0 == address(0)) {
        if (address(this).balance > 0)
            ct0.deposit{value: address(this).balance}(address(this).balance,
msg.sender);
```

```
    } else {
        uint256 amount0Collected =
IERC20Partial(token0).balanceOf(address(this));
        if (amount0Collected > 0) {
-           if (token0 != address(0))
            IERC20Partial(ct0.asset()).approve(address(ct0),
amount0Collected);
            ct0.deposit(amount0Collected, msg.sender);
        }
    }
```

Team response

Fixed.

Mitigation Review

The fix corrects the redundant query.

I-2 Array length mismatch would break migrations

Description

`migrateV3` and `migrateV4` do not validate the lengths of `swapAddresses`, `swapCalls`, If these arrays have mismatched lengths, it could lead to unexpected behavior or out-of-bounds errors.

Recommended mitigation

Add validation to ensure that the lengths of `swapAddresses`, `swapCalls`, and `swapValues` (if applicable) are equal.

```
function migrateV3(...) external {
    require(swapAddresses.length == swapCalls.length, "Mismatched array
lengths");
    // ..snip
}

function migrateV4(...) external {
    require(swapAddresses.length == swapCalls.length, "Mismatched array
lengths");
    // ..snip
}
```

I-3 Whitelist swapAddresses that can be called during batch swaps

Description

When migrating the logic allows for users to batch swap and then have the final `token0` and `token1` before depositing into the collateral tracker vaults.

```
        for (uint256 i = 0; i < swapAddresses.length; i++) {
            if (swapAddresses[i] == address(NFPM) || swapAddresses[i] ==
address(V4_PM))
                revert PeripheryErrors.InvalidSwapAddress();
```

As seen, during these batch swaps we use a completely unsafe external call, though the window of this attack vector has been reduced since even if we reenter via the call to `migrateV3()` or `migrateV4()`, only the owner of the tokenId is allowed to migrate.

Recommended mitigation

Whitelist the addresses on which users are allowed to swap to completely close the vector, as the call is now to a trusted address.

Team response

Acknowledged.

Mitigation Review

Acknowledged.

I-4 Incorrect Natspec/documentation

Description

In `migrateV4()`:

```
/// @notice Removes all liquidity from `tokenId` in the NFPM and deposits
into supplied `CollateralTracker` vaults.
/// @param tokenId The NFPM token ID to migrate.
```

Instances of NFPM should point to V4_PM instead since that's where we are migrating from.

Across the PanopticPool, in the new scope we now have a bool `usePremiaAsCollateral` which is documented as:

```
    /// @param usePremiaAsCollateral Whether to compute accumulated premia
for all legs held by the user for collateral (true), or just owed premia for
long legs (false)
```

However it should be *as* collateral and not *for*.

Recommended mitigation

```
- /// @notice Removes all liquidity from `tokenId` in the NFPM and deposits into supplied `CollateralTracker` vaults.
+ /// @notice Removes all liquidity from `tokenId` in the V4_PM and deposits into supplied `CollateralTracker` vaults.
- /// @param tokenId The NFPM token ID to migrate.
+ /// @param tokenId The V4_PM token ID to migrate.
```

```
-    /// @param usePremiaAsCollateral Whether to compute accumulated premia for all legs held by the user for collateral (true), or just owed premia for long legs (false)
+    /// @param usePremiaAsCollateral Whether to compute accumulated premia for all legs held by the user as collateral (true), or just owed premia for long legs (false)
```

Team response

Fixed.

Mitigation Review

Fixed as recommended.
