PREMIA SECURITY ASSESSMENT

August 17, 2023

Prepared By:

Arbitrary Execution

Changelog:

July 14, 2023	*Initial report delivered*
August 18, 2023	*Final report delivered*

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This report contains the results of Arbitrary Execution's (AE) assessment of the Premia V3 protocol. The protocol provides users with an on-chain, non-custodial options settlement engine with automated market maker functionality. The Premia V3 protocol allows users to create permissionless, European-style option Pools for arbitrary ERC20 token pairs, deposit liquidity into said Pools in exchange for a percentage of the fees generated as users interact with the Pool, and trade positions within those Pools. These option Pools expose both integration points for additional functionality, like the use of Vaults and quote systems, as well as utility functions such as using pool tokens to fund flash loans.

Four Arbitrary Execution engineers conducted this review over a 7-week period, from May 22, 2023 to July 7, 2022. The audited commit hash was `370d816f8d965509cc8535438e945298f9f59cf4` in the `Premian-Labs/premia-v3-contracts-private` repository. The Solidity files in scope for this audit consisted of a subset of the contract files in the `contracts` directory. Notably, contracts in the `orderbook`, `staking`, `utils`, `layerZero`, and `vendors` sub-directories were not included in the scope of the audit. The complete list of in-scope files is in Appendix B.

The team performed a detailed, manual review of each in-scope contract in the codebase. In addition to manual review, Slither was used for automated static analysis with custom detectors developed by AE.

The assessment resulted in findings ranging in severity from critical to note (informational). A critical finding in the way rewards are paid to users acting as Authorized Agents for position owners allows an arbitrary user to steal all the tokens held within any option Pool. This code pattern was repeated in three locations, creating three different avenues for attackers to exploit the vulnerability.

Two high severity findings impact the rebate claiming process for referrals and the use of Time-Weighted Average Price (TWAP) metrics as a security mechanism on the Arbitrum network.

The medium and low severity findings have a range of impacts on protocol functionality, including token price calculation, referral rebates, Vault updates, and Pool maturity date calculations.

The note severity findings contain observations regarding code hygiene, token ID generation, and erroneous checks.

FIX REVIEW UPDATE

The Premia team has fixed or acknowledged all major issues identified in the engagement. The full breakdown of fixes can be found in the Fixes Summary section.

Over the course of the engagement, the Premia team made a number of updates, including some bug fixes, to their protocol. As part of the fix review, the AE team also reviewed the pull requests containing those changes. These pull requests, along with AE's evaluation of each fix, are listed in Appendix C.

FIX REVIEW PROCESS

After receiving fixes for the findings shared with the Premia team, the AE team performed a review of each fix. Each pull request was scrutinized to ensure that the core issue was addressed, and that no regressions were introduced with the fix. A summary of each fix review can be found in the Update section for a finding. For findings that the Premia team chose not to address, the team's rationale is included in the update.

AUDIT OBJECTIVES

AE focuses on common, high-level goals for all security audit engagements. During this engagement, the AE team:

- Identified smart contract vulnerabilities
- Evaluated adherence to Solidity best practices

The Premia team provided an additional goal of evaluating the protocol's exposure to risk from external price oracle manipulation. To that end, the AE team spent additional effort reviewing the security mechanisms put in place to protect the Premia V3 protocol from such risk.

SYSTEM OVERVIEW

SYSTEM COMPONENTS

POOLS

Pools represent the core of the Premia V3 protocol. A Pool is implemented by a smart contract, which represents a European-style option market for a single ERC20 token pair. Each Pool is defined in terms of its maturity date, strike price, and option type (either Call or Put).

Over the lifetime of a Pool, Premia V3 users can both contribute liquidity to that Pool or trade options using the liquidity that already exists in the Pool. These two roles embody the traditional maker-taker model. Users that contribute liquidity to Pools are referred to as Liquidity Providers. Liquidity Providers fill the "maker" role, in that their deposited liquidity is used to fill the trades created by other users. When depositing liquidity, Liquidity Providers specify the range of prices (ticks) through which the deposited liquidity will be distributed within the Pool. The liquidity provided by Liquidity Providers can take the form of either token collateral or option contracts.

Users that trade options within a Pool are known as Liquidity Takers. Liquidity Takers fill the "taker" role by taking the liquidity in the Pool and using it to buy and sell option contracts. Option contracts are represented by fungible ERC1155 tokens. The use of these tokens allows option positions to be traded on third-party exchanges.

Unlike many other exchanges, Premia V3 uses a linear pricing system to derive market prices. Liquidity Providers must specify a price range for their deposit by specifying a lower and an upper tick value. Liquidity is then deposited linearly through that tick range. This distribution strategy ensures that market prices move linearly through tick ranges as the option market changes.

Trading within a Pool incurs transaction fees. These fees are split between Liquidity Providers and protocol stakeholders as a reward. These fees can be collected at any time, including before the maturity date of the Pool itself.

Pools also expose other, non-position related functionality to Premia V3 users. This set of functionality includes flash loans and an over-the-counter (OTC) quote system, which supports the creation of option quotes which are fillable by any Vault or market maker.

ADAPTERS

The Adapter contracts enable Premia V3 to retrieve prices for tokens. There are three main Adapter contracts that the protocol uses to retrieve price information:

- `ChainlinkAdapter`
- `UniswapV3Adapter`
- `UniswapV3ChainlinkAdapter`

The `ChainlinkAdapter` queries the Chainlink Price feeds on Arbitrum, the `UniswapV3Adapter` uses a Time-Weighted Average Price (TWAP) aggregated over relevant Uniswap V3 pools, and the `UniswapV3ChainlinkAdapter` is a composite contract that uses both Chainlink and Uniswap V3 TWAPs to determine price. Furthermore, the `UniswapV3ChainlinkAdapter` determines price by first getting a quote from the `UniswapV3Adapter` for the given `tokenIn/wrappedNative` pool and a quote from the `ChainlinkAdapter` for the given `wrappedNative/tokenOut` price feed and then multiplying the two quotes together to get a resulting `tokenIn/tokenOut` price (`tokenIn/tokenOut = tokenIn/wrappedNative * wrappedNative/tokenOut`).

There are a few major differences between the Chainlink and Uniswap V3 Adapter contracts:

1. The `ChainlinkAdapter` can provide both historical and spot prices, whereas the `UniswapV3Adapter` is best suited to providing a spot price where the TWAP is calculated using the past 10 minutes' worth of observations based on swaps in the relevant Uniswap V3 pools.
2. Anyone can attempt to add to the Adapters by calling `upsertPair`. However, the `ChainlinkAdapter` contract is access controlled for adding new price feeds, therefore prohibiting what token pairs can be supported. There is no access control in the `UniswapV3Adapter` for adding token pair paths. If there is a valid Uniswap V3 pool deployed for the given tokens, then anyone can add a token pair path to the `UniswapV3Adapter`.
3. The `UniswapV3Adapter` requires a Uniswap V3 pool for the exact token pair to retrieve price for, whereas the `ChainlinkAdapter` allows a single token "hop". The hop enables tokens that do not share a direct price feed to use an intermediate token shared by both tokens to calculate price.

All the Adapters will scale the resulting token price to 18 decimals.

VAULTS

Vaults are smart contracts that execute trading strategies on behalf of users. Vaults provide an alternative to personally managing positions in Pools, where by users deposit liquidity into Vaults, which then use their own strategies to interact with Pools. This workflow allows users to passively earn rewards for indirectly providing liquidity to Premia V3 Pools.

The Vault Registry contract handles storage for keeping track of each Vault that has been added to the Premia V3 protocol. Only the owner of the Vault Registry contract can perform administrative actions, such as adding and removing Vaults registered in the Vault Registry. Additionally, there are view-only methods that allow users to see what Vaults have been registered. These actions include, but are not limited to, viewing the Vaults by asset type, by trade side (Buy or Sell) and by option type (Call or Put). The Vault Registry has been designed in such a way that allows for many different types of Vaults to be compatible with the Premia V3 protocol, where each Vault contains its own strategy for interacting with the protocol. However, only the Underwriter Vault contract was reviewed within the scope of this audit.

The Underwriter Vault contract allows users to deposit assets to a Vault and receive a yield-bearing token in return. Each Vault accepts a single asset and is implemented using the ERC4626 standard. The Underwriter Vault is also implemented as a Proxy contract, where the implementation address of the Underwriter Vault is stored in the storage of the Vault Registry contract. As a Liquidity Provider of the Underwriter Vault, depositors can deposit a single asset (an ERC20 token) into the Vault, which will then be used as collateral to underwrite Call and Put options to buyers. As a result of depositing assets into the Vault, depositors will earn tokens in return, which accrue yield from the option premiums sold to buyers. In contrast, buyers can purchase options from the Vault by paying a premium so that they can receive long option contracts (represented as ERC1155 tokens).

When purchasing an option from the Underwriter Vault, the quoted price (premium) is equal to the fair option price calculated using the Black-Scholes model, plus spread and minting fees. The spread fee is determined by computing a C-Level, which is based on the utilization rate of the Vault's assets. As the utilization rate increases (i.e. more Vault assets are being used to underwrite options relative to the amount of assets in the Vault), so does the C-Level, and vice versa. This serves to incentivize or disincentivize traders to purchase options from the Vault based on the amount of assets in the Vault that are used, helping to balance the equilibrium of the Vault's liquidity and option prices. Overall, the Underwriter Vault contract is a layer built on top of Premia's V3 Pools and still requires interacting with the Pools to mint and allocate long option contracts to buyers.

POOL FACTORY

The Pool Factory is a smart contract through which users create new Pools. The Pool Factory's deployPool function deploys a new Pool contract on behalf of the caller, after which point other Premia V3 users can interact with the newly deployed Pool. The Pool Factory contract itself serves as the implementation of a separate proxy contract. The use of the proxy pattern supports an update process for the Pool Factory that does not impact user workflows.

REFERRALS

The `Referral` contract enables referrals for users and holds the accounting for tokens and their rebate amount for each referral. This contract is used in tandem when performing certain actions in Pools, such as trading, and the caller provides a referrer address. The `Referral` contract in turn transfers a percentage of the user's fee (trading fee, protocol fee, etc.), which is calculated based on the action performed, and credits that as a rebate to the referrer address. Referrer addresses are then able to claim their token rebates from the `Referral` contract at any time.

ROUTER

The Router is a smart contract that facilitates the transfer of ERC20 tokens. User interactions that involve sending ERC20 tokens to on-chain Premia V3 protocol components (like Pools) require users to approve the Router to perform token transfers on their behalf.

VOLATILITY ORACLE

The Volatility Oracle contracts are responsible for providing the Implied Volatility (IV) of an option contract, as well as the risk-free rate. The IV of a contract is calculated based the spot price of the token under contract, the strike price of the option, and the time until the option matures. The risk-free rate is a constant value based on the rate of return expected from a zero-risk investment.

USER CATEGORIES

LIQUIDITY PROVIDERS (MAKERS)

Liquidity Providers are responsible for adding liquidity to Pools. In the case of Call Pools, liquidity takes the form of ERC20 tokens. In the case of Put Pools, liquidity takes the form of long and short options (represented by ERC1155 tokens). In exchange for providing liquidity to Pools, Liquidity Providers receive a portion of the transaction fees that the Pool collects as Liquidity Takers interact with that Pool. Liquidity Providers can withdraw both their liquidity and collected fees from a Pool at any time.

LIQUIDITY TAKERS (TRADERS)

Liquidity Takers are users who buy and sell option contracts within a Pool. These trades are filled using the liquidity within a Pool (made available by Liquidity Providers). The Premia V3 protocol charges Liquidity Takers a fee, which is distributed to both Liquidity Providers and protocol stakeholders in the form of rewards, on every trade.

PRIVILEGED ROLES

PROXY CONTRACT OWNER

Premia V3 utilizes ownable proxy contracts to support on-chain upgradeability. Proxy Contract Owners are the only users allowed to update deployed proxy contracts to point to new implementation contracts.

POSITION OPERATORS

Position Operators are the only users allowed to modify existing positions within a pool. For example, only a position's Operator can transfer ownership of that position to another user.

AUTHORIZED AGENTS

Authorized Agents are users who are authorized to act on expired positions on behalf of the owners of those positions. As examples, an Authorized Agent can settle and exercise expired options. Authorized Agents receive a fee taken from the option's unlocked collateral in exchange for performing this service.

OBSERVATIONS

The Premia V3 protocol contracts are separated into logical components that contain NatSpec and other in-line comments. The code is well documented in some places, such as functions that operate on ticks. However, there are other places in the contract code, like in the Volatility Oracle, where complex operations are not documented.

The audited codebase contains a large Foundry unit test suite. This test suite includes both positive and negative unit tests that verify expected on-chain state changes and, in the case of negative unit tests, validate that the expected revert message was observed. The test suite relies on concrete input values when interacting with the Premia V3 API. As such, the current test suite could be improved by running tests with dynamic input values as well. Foundry's built-in fuzzing functionality provides one path for integrating tests that use dynamically generated input values.

During the audit, it was observed that the Premia V3 protocol is not compatible with tokens that may change holder balances (e.g., fee-on-transfer and rebasing tokens). This is because such balance changes are not accounted for in the internal accounting of the Premia V3 protocol. As a specific example, the `Referral` contract calculates and receives rebates from Pools, which are then credited to the respective referrer address. The exact rebate amount is stored, so that when a referrer attempts to claim a rebate that exact value can be transferred. However, if the rebated token amount has decreased either due to a fee-on-transfer structure or other negative rebasing action, this call can fail. In the worst case, this can prevent a referrer from being able to claim any rebates. This incompatibility is problematic because the Premia V3 protocol's permissionless Pool creation process does not restrict what tokens are used in Pools.

FINDINGS

CRITICAL SEVERITY

[C01] AUTHORIZED AGENTS CAN DRAIN LIQUIDITY FROM POOLS

Authorized Agents (AA) are users who are granted permission to perform privileged actions within a Pool on behalf of position holders within that Pool. The following functions reward AAs for performing such actions:

- `settleFor`
- `settlePositionsFor`
- `exerciseFor`

All these functions allow an AA to specify a batch of positions on which the Pool will perform some action. If the Pool does not revert while performing this action, then the AA will receive a payment, taken from the token balance that is owed to the owner of the position. However, each of the functions listed above contains a short-circuit path that causes the Pool to return before executing several security checks that revert the transaction. These paths allow an AA to supply malformed data to each of these functions but still receive payment taken from the Pool's own token balance. The following paragraphs describe how an attacker can use this code pattern to drain all the liquidity from a Pool. While this attack is specific to the `settlePositionFor` function, the other two functions can be exploited in a similar manner to the same effect.

The `settlePositionFor` function does not check the return value of calls to `_settlePosition`. Instead, `settlePositionFor` assumes that if none of the calls to `_settlePosition` revert, then all the positions in the batch were successfully settled. This assumption is then encoded in the calculation of the AA payout, in which the AA is paid a fee multiplied by the number of positions in the batch.

However, `_settlePosition` can return successfully before a position is settled, leaving that position unsettled. An attacker can cause `_settlePosition` to return early by calling `settlePositionFor` with a crafted `Position.Key` struct. Crafting such a struct only requires setting one of the fields to a value for which there is no existing position. The result is that `_settlePosition` will generate an invalid `tokenId`, which will produce a zero value when used to query a user's balance. This zero value then triggers the early return from `_settlePosition`. The `settlePositionFor` function then proceeds to pay the AA from the Pool's token balance.

An attacker can also control the value of the fee they receive for settling a position. The `setAuthorizedAgents` function allows an attacker to set themselves as their own AA. Doing so allows the attacker to pass the agent check in `settlePositionFor`. The `setAuthorizedCost` function allows an attacker to set the size of the fee that an AA (the attacker in this case) receives for settling a position. Setting this value allows the attacker to bypass any upper bound on the payout an AA receives for settling a position.

By combining these two primitives (triggering the fee payout and setting the size of the fee), an attacker can drain all the liquidity from an arbitrary Pool. The first step is to query the number of pool tokens held by the Pool and set the AA's fee to that amount. With the AA's fee set, the attacker can then trigger the fee payout with the crafted `Position.Key` struct, resulting in the attacker receiving all the pool tokens held by the Pool.

RECOMMENDATION

Consider updating `_settlePosition`, `_settleFor`, and `_exercise` to return an additional `bool` value corresponding to whether the position was settled successfully, and also updating the calling functions (`settlePositionFor`, `settleFor`, and `exerciseFor`) to only pay the AA for the number of successfully completed actions.

UPDATE

Fixed in pull request #252 (commit hash `ea5743508039fff1b3e58f18b7128b6185ecff6e`) as recommended, with the minor change that an unsuccessful settlement causes the transaction to revert.

HIGH SEVERITY

[H01] INVALID UNISWAP V3 TWAP SECURITY ASSUMPTIONS

Time-Weighted Average Price (TWAP) is used in the `UniswapV3Adapter` contract for price quotes. The `UniswapV3Adapter` contract can either be used for standalone price quotes, or for aggregate price quotes as seen in the `UniswapV3ChainlinkAdapter` contract. TWAP uses the `observations` stored for a Uniswap V3 pool to calculate a price that takes into consideration each stored price, the amount of time that price was held for, and the cumulative liquidity. This means a TWAP will be more resistant to large changes in a Uniswap V3 pool that only last for a short duration, whereas a spot price would be more volatile. Additionally, the number of observations a Uniswap V3 pool stores would increase the cost of an attack, as an attacker would now be required to hold a manipulated price for some duration of the block window used for a given TWAP calculation. An attacker could try to manipulate a TWAP with a single, large price change, but this type of attack against a TWAP with a large block time range would cost an exponential amount of money. Therefore, the primary way for an attacker to manipulate a TWAP would be to hold a manipulated price for a Uniswap V3 pool over a period of several blocks.

On Ethereum Mainnet, the TWAP of a Uniswap V3 pool is implicitly protected against a multi-block price manipulation attack due to Maximal Extractable Value (MEV) activity. MEV dissuades attackers from holding a manipulated price in a Uniswap V3 pool for over a block because MEV bots would simply arbitrage the price back to the original price tick, and take the liquidity of the attacker in the same block that the attack was launched in. On Arbitrum, this implicit protection does not exist because Arbitrum's Sequencer operates differently than Ethereum Mainnet's Proof-of-Stake. Arbitrum's Sequencer uses a private mempool, and transactions are ordered on a First Come First Serve (FCFS) basis. This in turn means that MEV searchers on Arbitrum cannot see transactions before they are included in a block, and even if they could see transactions before they are included, it would not matter as it would be impossible to change the ordering of transactions that have already been received by the Sequencer. This removes the cost of arbitrage from an attack on a Uniswap V3 pool's TWAP.

Additionally, Arbitrum's Sequencer handles transactions quickly enough where, in most cases, only one transaction is processed per block (multiple transactions per block may occur under heavier network traffic). As a result, this allows an attacker to create two separate transactions: one swap to move a target Uniswap V3 pool into the target price tick, and another swap to move the price tick back (self-arbitrage to avoid losses). These transactions will be processed one after another and, assuming normal network conditions, will be placed into separate blocks. If the two blocks have different timestamps (Arbitrum can process more than one block a second), then the second swap transaction will write the price of the first transaction into the `observations` array of the Uniswap V3 pool. Assuming that timing is the only issue an attacker will have to handle, and with the cost of gas on Arbitrum being low, an attacker should be able to reliably write price observations into a Uniswap V3 pool.

Therefore, based on the assumptions presented above, an attacker can perform any number of "two-block" TWAP attacks. The cost associated with performing multiple two-block TWAP attacks is equal to the amount of liquidity needed to move a Uniswap V3 pool to the desired price tick multiplied by the pool's fee tier and the total number of swaps made. While this cost may not be small, it is significantly less than the cost of arbitrage and the exponential cost of moving price ticks over a single block.

RECOMMENDATION

The protection against price manipulation attacks when using the Uniswap V3 TWAP for a given Pool can be increased by the following measures:

1. Ensuring consistent liquidity across a broad range of price ticks, which prevents attackers from easily changing the price tick with a relatively small amount of liquidity.
2. Ensuring frequent trade activity such that the observation window used for a TWAP calculation is fully populated. This prevents an attacker from being able to manipulate the TWAP using a smaller number of trades.

However, Premia V3's current architecture allows any user to create new Pools with any valid Uniswap V3 pool. This makes it difficult to guarantee that the above conditions will hold, as users would be expected to perform these actions, which is cumbersome and potentially expensive. Additionally, a dedicated attacker may still be able to beat out normal users in writing price observations by decreasing their network latency to the Arbitrum Sequencer.

Due to the complexities described above, consider removing Uniswap V3 TWAP as a price oracle on Arbitrum.

UPDATE

Fixed in pull request #345 (commit hash `ffdb7622a63075e9f506180d38e62cb6c119046a`) with the removal of the `UniswapV3Adapter` and `UniswapV3ChainlinkAdapter` contracts.

[H02] RE-ENTRANCY IN THE `CLAIMREBATE` FUNCTION ALLOWS TOKENS TO BE STOLEN FROM THE `REFERRAL` CONTRACT

The `claimRebate` function in the `Referral` contract enables users to withdraw the rebates they are owed. This is accomplished by iterating over each token a user has rebates for, and transferring the stored rebate amount to the user. Before transferring the rebate amount, the function follows the checks-effects-interactions pattern by removing the token from the rebates data structure and setting the rebate amount to zero. This attempts to prevent an attacker from claiming more tokens than they should by re-entering into `claimRebate`.

However, this alone is insufficient to protect against re-entrancy attacks, and the function lacks a re-entrancy guard. Therefore, if the function is re-entered into, the call to `getRebates` will return the tokens that have not yet had their rebates claimed. This enables an attacker to duplicate later tokens in the array and claim rebates on them more than once. Using this attack vector, an attacker could create many Pools with custom re-entrant tokens, and trade with them to generate referral rebates that could then be used to duplicate other token rebates multiple times. In the worst case, this could be used to drain the `Referral` contract of all its tokens. Additionally, stealing tokens from the `Referral` contract creates a deficit that will prevent users from being able to collect any subsequent rebates generated for the tokens that were stolen.

The following outlines a possible attack scenario:

1. The `Referral` contract holds tokens that will be distributed to various users.
2. An attacker, Alice, creates malicious ERC20 tokens that are re-entrant to exploit the `claimRebate` function.
3. Alice proceeds to create many Pools via the `PoolFactory` contract for the re-entrant ERC20 tokens.
4. Alice creates a new contract which will be the referrer contract and will trigger the re-entrancy.
5. Alice creates trades against the new Pools and the target Pool to steal tokens from, where the attack contract is the referrer. Alice orders the trades such that the target token to steal is duplicated as many times as possible, and each token (re-entrant and the target) is added as a rebate token to the attacker contract.
6. The attacker contract calls `claimRebate`, and each time one of the re-entrant tokens is transferred it calls out to the attacker contract, which will re-enter into `claimRebate`. This duplicates the array of tokens and rebates that will be collected, thus enabling the attacker contract to claim rebates on tokens more than once.
7. The amount of tokens Alice and her attacker contract can steal is dependent on the number of re-entrant tokens the attacker contract can claim a rebate on.

RECOMMENDATION

Consider adding a re-entrancy guard to the `claimRebate` function.

UPDATE

Fixed in pull request #261 (commit hash `0552dea9c52ce66b05341e31278379b585d75456`), as recommended.

MEDIUM SEVERITY

[M01] CREATING NEW UNISWAP V3 POOLS FOR FEE TIERS CAN DOS PRICE QUOTES

When requesting a quote by calling the `quote` or `quoteFrom` functions in the `UniswapV3Adapter` contract, there is a check to ensure a Uniswap V3 pool's cardinality is greater than or equal to the target cardinality required by the `UniswapV3Adapter` contract. If this check fails, the transaction will revert. The `quote` and `quoteFrom` functions can experience a temporary DoS if an attacker creates new token pair pools for fee tiers that did not already exist. This can happen if a new Uniswap V3 pool is created for a token pair that has already been registered with the `UniswapV3Adapter`, and the desired target cardinality is not set to a value greater than or equal to the target cardinality required by the `UniswapV3Adapter`.

RECOMMENDATION

Consider ensuring that the `upsertPair` function on the `UniswapV3Adapter` contract is always called before requesting a quote from the `quote` and `quoteFrom` functions. This would ensure that the desired target cardinality on the Uniswap V3 pool is increased to equal the target cardinality required by the `UniswapV3Adapter` contract before requesting a quote.

UPDATE

Fixed in pull request #345 (commit hash `ffdb7622a63075e9f506180d38e62cb6c119046a`) with the removal of the `UniswapV3Adapter` and `UniswapV3ChainlinkAdapter` contracts.

[M02] FETCHING HISTORICAL PRICE DATA WITH CHAINLINK MAY FAIL IF THE UNDERLYING AGGREGATOR IS UPGRADED

The `_fetchQuoteFrom` function attempts to retrieve the historical price of `tokenIn` denominated in `tokenOut` at the passed-in `target` timestamp by looping until either the `aggregatorRoundId` reaches zero or a price is found near the requested timestamp. However, the `phaseId` value is never decremented. Per the Chainlink docs, `phaseId` indicates the number of aggregator contracts that have existed for the given price feed. Each aggregator holds its own historical price which means that the most recent `phaseId` (and by extension the most recent underlying aggregator contract) may not have the price at the requested timestamp, but that does not mean that there does not exist a price at or near the requested timestamp.

Therefore, the underlying `_fetchQuoteFrom` function will only be able to retrieve historical prices for, at the earliest, the minimum timestamp returned by `aggregatorRoundId = 1` for the given `phaseId`. This means that if the underlying aggregator contract was changed, then `_fetchQuoteFrom` may start to revert for historical price data that it may have been able to retrieve before the update but cannot retrieve post-update.

RECOMMENDATION

Consider adding logic to decrement `phaseId` (if greater than one) when `aggregatorRoundId = 1`, which indicates that the end of the historical price data for the current aggregator has been found. This enables `_fetchQuoteFrom` to continue searching for historical price data at the passed-in timestamp.

UPDATE

Fixed in pull requests #386 (commit hash `a28610a38a6f81cdc57b16eed1d6c0fbde74735b`), #400 (commit hash `904fa61879e77447f7b0da657d29791208179f4d`), #402 (commit hash `006d8537e83d72b9db22ea894fcdc8358f25785f`), and #411 (commit hash `1d41da618f1f0dd265b1e55d91b41eb64f04bf50`). Due to unexpected issues with the historic price data returned by Chainlink aggregators, the Premia team opted to add functionality to the Chainlink Adapter to manually push price updates. This allows Adapters to provide historical pricing data in the event that the data is either not available from the current Chainlink aggregator or it is available but deemed stale.

[M03] COLLECTING REFERRAL REBATES IS VULNERABLE TO DOS ATTACKS BY MALICIOUS TOKENS

The `claimRebate` function in the `Referral` contract allows users to claim rebates they have received from referrals. This is accomplished by iterating over all the tokens that the referrer has rewards for, and transferring the specified token amount to them. However, if any transfer call were to fail inside the loop, then the function will fail, preventing users from claiming their rebates.

Therefore, an attacker could create a malicious token designed to DoS the `claimRebate` function by failing only when the `Referral` contract attempts to transfer the token out, thus preventing the `claimRebate` function from working for any user that has a non-zero rebate amount for the malicious token. The following example outlines the attack scenario:

1. An attacker creates a new ERC20 token that reverts when `transfer` is called by the `Referral` contract.
2. The attacker creates a new Pool in the Premia protocol with the malicious token.
3. The attacker then distributes the malicious token to various addresses to create trades where the referrer addresses are the target DoS victims. By forcing the malicious token to be used as a rebate to a victim address, the attacker can deny the victim from claiming any rebates, including rebates from other Pools.

RECOMMENDATION

Consider updating `claimRebate` to take in a user-specified list of rebate token addresses. This allows the user to exclude malicious tokens and still claim the rebates from the tokens they want.

UPDATE

Fixed in pull request #260 (commit hash `4f2605bf31f50b0363d7d8a0f784aa5c3400187f`), as recommended.

[M04] THE `OBSERVATIONCARDINALITY` FOR A UNISWAP V3 POOL MAY BE INSUFFICIENT FOR PRICE QUOTES

When adding a Uniswap V3 pool to the `UniswapV3Adapter` contract via the `upsertPair` function, the cardinality of the pool is increased so that the pool matches the expected TWAP time period set in the contract. However, this does not guarantee that the actual cardinality of the Uniswap V3 pool is the expected cardinality set during `upsertPair`. This is because cardinality is only updated when a new observation is written to the pool. Observations are only written to a pool when a swap occurs and a previous observation has not already been written for the same `block.timestamp`.

Additionally, when retrieving a TWAP for a given time interval, there are no checks in `_calculateRange` to ensure that the `observationCardinality` is greater than or equal to the expected cardinality. Therefore, it is possible for a Uniswap V3 pool to have a single observation that is old enough to satisfy the TWAP time period and thus return that single observation as the price. This effectively reduces the TWAP to a spot price, which is dangerous for Uniswap V3 pools as they are extremely easy to manipulate for a single observation.

This is also an issue for the Composite Adapter, which uses the Uniswap V3 Adapter to partially derive price information.

RECOMMENDATION

Consider checking that the `observationCardinality` is not less than the `observationCardinalityNext` stored in `slot0` for a Uniswap V3 pool when retrieving token price via `quote` or `quoteFrom` in the `UniswapV3Adapter` contract.

UPDATE

Fixed in pull request #345 (commit hash `ffdb7622a63075e9f506180d38e62cb6c119046a`) with the removal of the `UniswapV3Adapter` and `UniswapV3ChainlinkAdapter` contracts.

LOW SEVERITY

[L01] USERS COULD EXPERIENCE UNEXPECTEDLY HIGH GAS FEES

The `_fetchQuoteFrom` function attempts to retrieve historical price data by iterating backwards through all possible values of `aggregatorRoundId` until it finds a price near the specified `target` timestamp. This type of search scales linearly with respect to the distance between the `target` timestamp and the `updatedAt` timestamp of the most recent round. As a result, this algorithm performs inefficiently when dealing with a large delta between timestamps. As mentioned in the Chainlink docs, looping for historical price data can be very expensive to do on-chain.

As an example of the potential cost of doing a linear search, consider the `ETH / USD` price feed on Arbitrum. As of the current time, calling `latestRoundData` returns a `roundId` of `18446744073709953264` with an `updatedAt` UNIX timestamp of `1686853712`. To retrieve a historical timestamp of approximately 24 hours ago from the current (as of this time in writing) timestamp, it would take approximately 584 iterations (which would result in a `roundId = 18446744073709952680`).

While the cost of gas on Arbitrum is less expensive than Ethereum Mainnet, iterating hundreds of times could still generate a high cost in gas fees. Additionally, iterating too many times could cause the transaction to hit Arbitrum's soft gas limit of 32M.

RECOMMENDATION

Consider using a binary search algorithm to fetch historical price from Chainlink, which is logarithmic with respect to the distance between the `target` timestamp and the `updatedAt` timestamp for the most recent round.

UPDATE

Fixed in pull request #294 (commit hash `e503eb439c0e4c5ef15af36f9d3321dcf6fbfba7`), as recommended.

[L02] SUDDEN PRICE CHANGES MAY RESULT IN INACCURATE HISTORICAL PRICES

The `_fetchQuoteFrom` function in the `ChainlinkAdapter` contract attempts to fetch a historical price at or near the passed-in `target` timestamp. If a price at the exact timestamp cannot be found, `_fetchQuoteFrom` will determine the closest price based on the round timestamp and return that instead. However, taking the price that is closest to the `target` timestamp may not account for sudden price changes between the two rounds. This could cause the Adapter to report an inaccurate price.

RECOMMENDATION

Consider taking the mean of the two prices whose round timestamps bound the `target` timestamp, assuming the `target` timestamp does not equal either bound. This can be accomplished by taking the price of the left-bound round plus the delta price of both rounds (where the delta can be positive or negative) that bound the `target` timestamp. Taking the mean between two timestamps is similar to what Unsiwap V3 does. Additionally, the timestamp returned for this calculated mean price can simply be the `target` timestamp. However, the logic should take care to avoid the following conditions:

1. A price found for the passed-in `target` timestamp has an `updatedAt` timestamp that is earlier than `target`.
2. A round has not yet been created such that its `updatedAt` timestamp is greater than the `target` timestamp.

If the above conditions are met, then the function should still revert if the price is deemed to be stale. This is because the right-bound round has not been submitted to Chainlink, and thus a mean price cannot be calculated yet.

UPDATE

Acknowledged. The Premia team’s statement for this issue:

> *We won’t be implementing the changes recommended here as events happening after maturity should not affect the settlement price. It would also make us vulnerable for example to a scenario where a token price moves so suddenly after maturity, that the update published by Chainlink is much beyond the 0.5% deviation threshold, making the price given by the mean of the two observations worse than the one from last update before maturity, which would be at worst, 0.5% off. It also would mean that users would have to wait up to 24h post maturity before being able to exercise/settle, which would be less than ideal.*

[L03] USE OF INVALID `POSITIONS` MAPPING ENTRIES

All valid positions are stored in the `positions` mapping in the `PoolStorage` contract. The following functions use this storage, but do not directly test that they have retrieved a `Data` struct instance and not the default value at a mapping entry to which a value has never been written:

- `_claim`
- `_withdraw`
- `_deposit`
- `_transferPosition`
- `_settlePosition`

The use of the default value retrieved from a mapping could result in unexpected behavior while executing any of the above listed functions.

RECOMMENDATION

Consider adding an `isValid` field to the `Data` struct and setting that field to `true` when a position is created. This field can then be checked after each lookup into the `positions` mapping to ensure that the returned `Data` struct is valid. Alternatively, consider checking whether the address stored in the `owner` field of the `Key` struct has a non-zero balance for the `tokenId` of the corresponding position. The `_withdraw` function already performs this indirect check on the validity of the returned `Data` struct.

UPDATE

Fixed in pull requests #372 (commit hash `5d3549d7176dc5f36b1a59f8c4da9343a95dcd9b`) and #382 (commit hash `cef654193c1324cfefd5e99dd4918b17b00ab8e7`), as recommended.

[L04] OFF-BY-ONE ERROR IN `ISLASTFRIDAY` FUNCTION

There is an off-by-one error in the `isLastFriday` function in the `OptionMath` contract. This function returns a boolean value indicating whether or not a given timestamp falls on the last Friday of the month. The function first calls `DateTime.getDay` and `DateTime.getDaysInMonth` on `maturity` to find the day the timestamp falls on and the last day of the month where the timestamp falls. The off-by-one error occurs during the conditional statement in the following code snippet:

```
if (lastDayOfMonth - dayOfMonth > 7) return false;
return isFriday(maturity);
```

The conditional returns `false` for any date that isn't in the last week of the month, which should eliminate the need to check if that date is a Friday. There is, however, an edge condition that causes this logic to fail. If the month ends on a Friday and `dayOfMonth` is the previous Friday, `lastDayOfMonth - dayOfMonth == 7`, which passes this check. Then, when `isFriday` is called on the maturity date, the function will return `true`.

While no security risks to the protocol were identified because of this issue, it still represents a bypass of a high-level protocol invariant.

RECOMMENDATION

Consider updating the conditional statement described above to remove the edge case. This can be accomplished with the following check:

```
if (lastDayOfMonth - dayOfMonth >= 7) return false;
```

UPDATE

Fixed in pull request #303 (commit hash `081966a04ffa085629e48f9376e03df5f5cd727d`), as recommended.

[L05] LOCKED NATIVE TOKENS

The `deployPool` function is `payable` so that callers can send the network's native tokens required to pay the Pool's initialization fee. If the initialization fee is greater than zero, `deployPool` checks if the caller overpaid this fee and refunds any excess. If the initialization fee is zero, but a user sends tokens to the `PoolFactory` contract via the `deployPool` function, the user will not be refunded those tokens. Since the `PoolFactory` contract does not contain a function for transferring tokens, any tokens remaining in its balance after a transaction completes are locked into the contract and cannot be removed.

The impact of this issue is limited due to the fact that the `PoolFactory` contract is deployed using an upgradeable proxy pattern. As such, the `PoolFactory` implementation could be updated later to include a means of transferring locked tokens out of the `PoolFactory`.

RECOMMENDATION

Consider updating the `deployPool` function to always refund callers who overpay the initialization fee, regardless of the value of that fee.

UPDATE

Fixed in pull request #353 (commit hash `f1bb364ad714743cf06df132bf74272562625c2f`). The `deployPool` function was updated to revert if the Pool's initialization fee is zero.

[L06] POOL MATURITY TIME CHECK DOES NOT MATCH WHITEPAPER

According to the Premia V3 whitepaper, all Pools must expire at 8 AM UTC:

> All newly created option markets must expire at 8 AM UTC, with the additional stipulation that options with maturities over 2 days must expire on a Friday, and options with maturities over 30 days must expire on the last Friday of the calendar month.

The “8 AM UTC” check is performed when a Pool is deployed by the `_revertIfOptionMaturityInvalid` function:

```
if ((maturity % 24 hours) % 8 hours != 0) revert
PoolFactory__OptionMaturityNot8UTC(maturity);
```

However, this check will pass any timestamp that is evenly divisible by eight hours. The set of timestamps that will pass this check consists of 8 AM UTC, 4 PM UTC and 12 AM UTC. This allows a user to bypass the 8 AM UTC expiry requirement by setting their Pool to expire at either 4 PM UTC or 12 AM UTC. While no security risks to the protocol were identified because of this issue, it still represents a bypass of a high-level protocol invariant.

RECOMMENDATION

Consider updating the check in `_revertIfOptionMaturityInvalid` to restrict the set of valid maturity timestamps to only include 8 AM UTC.

UPDATE

Fixed in pull request #302 (commit hash `4124298b9bb4ec34fa8bcfb3ca4647500b574773`), as recommended.

[L07] THE `QUOTE` FUNCTION IN THE `CHAINLINKADAPTER` CAN RETURN STALE PRICE DATA

Under normal operational conditions, it is assumed that Chainlink operators will continuously provide up-to-date price data. Therefore, the underlying `_fetchLatestQuote` function called as part of the `quote` function in the `ChainlinkAdapter` contract does not check whether the returned price quote is stale. However, this is a dangerous assumption, as circumstances outside of Chainlink's control can prevent aggregators from returning up-to-date price data. For example, the Arbitrum Sequencer recently experienced service degradation that lasted several hours. This impacted the ability for transactions to be correctly processed and batched, which could include Chainlink aggregator price updates. This in turn could cause stale price data to be returned, up until the point at which the Chainlink aggregators are able to successfully submit price updates on-chain.

RECOMMENDATION

Consider adding a maximum time delta allowed for the most recent round, and if the round's `updatedAt` timestamp is older than the delta, then revert.

UPDATE

Fixed in pull request #361 (commit hash `05736a6eb8828b69e250f277987b831735c66db5`), as recommended.

[L08] UPDATEVAULT FAILS TO UPDATE VAULTSBYOPTIONTYPE AND VAULTSBYTRADESIDE IN VAULT REGISTRY

The updateVault function in the VaultRegistry contract is used to update properties of an existing Vault that has already been added to the Vault Registry. The issue exists when an owner calls the updateVault function to update the OptionType of a Vault from Both to either Call or Put. The same issue also exists when an owner updates the TradeSide of a Vault from Both to either Buy or Sell. When the updateVault function is called and either of the aforementioned conditions hold true, the newly updated Vault will not be added to the correct mapping and the existing Vault will be removed from that mapping.

This issue can be demonstrated by updating the OptionType of an existing Vault. When the OptionType of a Vault is updated from type Both to either Call or Put, there is logic to add the updated Vault into the vaultsByOptionType mapping. However, this logic does not work because the SolidState EnumerableSet contract will not add duplicate items:

```
function _add(
    Set storage set,
    bytes32 value
) private returns (bool status) {
    if (!_contains(set, value)) {
        set._values.push(value);
        set._indexes[value] = set._values.length;
        status = true;
    }
}
```

Because there is already an entry for a Vault for the Call and Put option keys in vaultsByOptionType, there will still only be one entry (the old entry) instead of the two. Then, the logic that is supposed to remove the old entry and keep the new entry will remove the one remaining entry. As a result, the new entry is not added to the mapping and the old entry is removed from the mapping.

RECOMMENDATION

Consider moving the logic to add an updated entry to both the vaultsByOptionType and vaultsByTradeSide mappings to after the logic that removes the existing Vault entry in those mappings. By doing so, the old entry will be removed before adding the newly updated entry to the EnumerableSet.

UPDATE

Fixed in pull request #358 (commit hash 90db1143b5a36331dfec64c32f0198964981f0a2), as recommended.

[L09] `UPSERTPAIR` MAY FAIL TO UPDATE TOKEN PAIRS

The `upsertPair` function in the `UniswapV3ChainlinkAdapter` contract has incorrect checks that can cause the function to not update the passed-in token pairs, or even revert. The following checks are incorrect under certain conditions:

- The `!isCached` check against the `UniswapV3Adapter` causes the call to skip calling `upsertPair` on the underlying Adapter if there exists at least one cached Uniswap V3 pool for the token pair. However, there can be multiple Uniswap V3 pools for a token pair (one for each fee tier) that may be created after the token pair is registered with the Adapter. This can cause the function call to erroneously not update the token pair even though there may be new Uniswap V3 pools to register.
- The `!hasPath` check against the `ChainlinkAdapter` causes the call to revert if a price feed is not found for the token pair. However, if a price feed is removed via the `batchRegisterFeedMappings` function then `upsertPair` needs to be called in order to remove the cached `PricingPath`. This check prevents `upsertPair` from being called because `hasPath` will be false when the price feed is removed.

RECOMMENDATION

Consider implementing one of the following suggestions:

1. Removing both the `isCached` check for the `UniswapV3Adapter` and the `hasPath` check for the `ChainlinkAdapter`.
2. Removing all `isPairSupported` checks from the `UniswapV3ChainlinkAdapter` contract, and simply calling `upsertPair` on the underlying `UniswapV3Adapter` and `ChainlinkAdapter` contracts. Each `upsertPair` function in the underlying contracts has its own sanity checks that are sufficient.

UPDATE

Fixed in pull request #345 (commit hash `ffdb7622a63075e9f506180d38e62cb6c119046a`) with the removal of the `UniswapV3Adapter` and `UniswapV3ChainlinkAdapter` contracts.

[L10] INCORRECT DOCUMENTATION

The following lines contain incorrect documentation:

- In `PoolFactory.sol`:
  - Line 171: “0” should be “255”
- In `IVaultRegistry.sol`:
  - Line 165: “implementation” should be “settings”
- In `UnderwriterVaultProxy.sol`:
  - Line 21: “Errors” comment is not relevant to surrounding code

RECOMMENDATION

Consider making the suggested changes to fix the documentation errors.

UPDATE

Fixed in pull requests #229 (commit hash `15ecc57a713682b28d847b24acfc86d9f0ee1e2d`) and #350 (commit hash `d57e7d005a914ff625c65a486fd062bfe5bdc7ef`), as recommended.

[L11] MISSING EVENTS FOR IMPORTANT STATE CHANGES

The following functions result in important state changes, but do not emit events:

- `insertFeeTier` in `UniswapV3Adapter.sol`

Events for important state changes (e.g. changing owner) should be emitted for off-chain tracking.

RECOMMENDATION

Consider adding additional events for key state changes.

UPDATE

Fixed in pull request #345 (commit hash `ffdb7622a63075e9f506180d38e62cb6c119046a`) with the removal of the `UniswapV3Adapter` and `UniswapV3ChainlinkAdapter` contracts.

[L12] USERS CAN MAKE THEMSELVES THE PRIMARY AND SECONDARY REFERRAL ADDRESS

The `Referral` contract enables users to receive referral rebates from Pools, where the calculated rebate is a percentage of the trading/protocol fee taken for a given action against a Pool. However, a user can create a separate address and trade against it with their address as the referrer to reap both the primary and secondary rebates on a trade, thus allowing a user to recoup a portion of the calculated fee.

RECOMMENDATION

Consider disallowing the primary and secondary referrer address from being the same address as the caller. Additionally, consider creating a new base referral tier that gives a 0% referral percentage to deter users from making separate addresses to receive some percentage of the fee back during Pool function calls.

UPDATE

Acknowledged. The Premia team's statement for this issue:

> *The conclusion was that it doesn't really make sense to guard against it because a user can make multiple accounts that they control. Also by default our frontend will set a default referral address controlled by us in case the user didn't come through another ref link, which will prevent him to refer himself afterwards. (Though sure he could create a secondary account). This is why the first referral tier will be 5% which we consider low, as there is no real way to solve this. This follows the approach taken by GMX.*

NOTE SEVERITY

[N01] EXTRANEOUS USE OF BOOLEAN LITERALS

The following lines of code perform a comparison with a boolean literal in the condition of an `if` statement:

- `safeTransferFrom` in `ERC20Router.sol`
- `writeFrom` in `PoolInternal.sol`

These statements can be simplified by removing the comparison and checking the value in the condition directly.

RECOMMENDATION

Consider removing these comparisons and instead checking the value directly in the condition. Consider using the `!` operator to check for falsity.

UPDATE

Fixed in pull request #370 (commit hash `f32cb6aaccf524996b9a26eb57f34a81073ea31f`), as recommended.

[N02] USE OF FLOATING COMPILER VERSION PRAGMA

All contracts in this repository float their Solidity compiler versions (e.g. `pragma solidity >=0.8.19`).

Locking the compiler version prevents accidentally deploying the contracts with a different version than what was used for testing. The current pragma prevents contracts from being deployed with an outdated compiler version, but still allows contracts to be deployed with newer compiler versions that may have higher risks of undiscovered bugs.

It is best practice to deploy contracts with the same compiler version that is used during testing and development.

RECOMMENDATION

Consider locking the compiler pragma to the specific version of the Solidity compiler used during testing and development.

UPDATE

Fixed in pull request #384 (commit hash `39b87f178659ecd4671f0ca5f539b1fad03013da`). The Premia team locked the compiler pragmas in their contracts to a specific version of the Solidity compiler and locked the pragmas in their libraries and interfaces to the current minor version of the Solidity compiler.

[N03] CHAINLINKADAPTER CAN RETURN CACHED PATHS FOR INVALID TOKEN PAIRS

The `isPairSupported` function in the `ChainlinkAdapter` contract will return `(true, true)` for token pairs whose price feed has been removed from the `FeedRegistry` but have not yet had `upsertPair` called. This is because the `batchRegisterFeedMappings` function does not remove the cached `PricingPath` value. This in turn causes `isPairSupported` to short-circuit and return early when `_pricingPath` returns a `PricingPath` enum that is not `NONE`.

RECOMMENDATION

Consider marking `batchRegisterFeedMappings` as `virtual`, and overriding the function in the `ChainlinkAdapter` contract such that `upsertPair` is called once the feed registry is updated.

UPDATE

Fixed in pull requests #364 (commit hash `a8e0ec11a356e89ad69a4d778ec0afefdddc7ade`) and #379 (commit hash `7e14a0c39789ff36dc385f8547ac50a5062d77cb`). Token pairings are now tracked in the `pairedTokens` mapping. This mapping is used in `batchRegisterFeedMappings` to set the cached `PricingPath` value to `NONE` for all tokens paired with a token whose price feed was removed.

[N04] USE OF MAGIC VALUES

The following values are used in the `UniswapV3AdapterProxy` contract, but neither their significance nor how they were calculated are documented:

- `100`
- `500`
- `3_000`
- `10_000`

The lack of documentation obscures the developers' intent in choosing these values.

RECOMMENDATION

Consider adding a comment to each undocumented value explaining its use and, if applicable, how it was calculated.

UPDATE

Fixed in pull request #345 (commit hash `ffdb7622a63075e9f506180d38e62cb6c119046a`) with the removal of the `UniswapV3Adapter` and `UniswapV3ChainlinkAdapter` contracts.

[N05] FORMATTOKENID OVERFLOWS MULTIPLE FIELDS IN GENERATED TOKEN IDS

The `formatTokenId` function generates a token ID by storing values at different offsets within a `uint256` value:

```
function formatTokenId(
    address operator,
    UD60x18 lower,
    UD60x18 upper,
    Position.OrderType orderType) internal pure returns (uint256 tokenId) {
        tokenId =
            (uint256(TOKEN_VERSION) << 252) +
            (uint256(orderType) << 180) +
            (uint256(uint160(operator)) << 20) +
            ((upper.unwrap() / MIN_TICK_DISTANCE.unwrap()) << 10) +
            (lower.unwrap() / MIN_TICK_DISTANCE.unwrap());
    }
```

The `formatTokenId` function reserves ten bits for the values stored in the `upper` and `lower` parameters. However, both parameters are of type `UD60x18`, which has a size of 256 bits when unwrapped. This means that any `upper` or `lower` value greater than `2 ** 10 - 1` (after accounting for `UD60x18` unwrapping and the `MIN_TICK_DISTANCE` division) will overflow the 10 bits reserved for that value, corrupting the values before it in `tokenId`.

The `formatTokenId` function is called by the following functions:

- `_deposit`
- `_withdraw`
- `_pendingClaimableFees`
- `_transferPosition`
- `_settlePosition`
- `_claim`

Although corruption can occur, all code paths through which corruption is possible eventually catch the malicious `upper` or `lower` value and trigger a revert. Both `_deposit` and `_withdraw` catch attempts to pass in a value large enough to trigger the corruption in calls to `_revertIfRangeInvalid` before the corruption occurs. The `_pendingClaimableFees` function also catches malicious `upper` and `lower` values in the calls to `_getTick`. In the case of `_transferPosition`, an attacker can trigger the corruption, but the overflowing `upper` or `lower` value is eventually detected in the subsequent `_pendingClaimableFees` call. Similarly, the overflow occurs in `_settlePosition`, but the responsible `upper` or `lower` value is detected in later calls to `_getTick`. In the case of `_claim`, the corruption occurs but is later detected by a call to `_pendingClaimableFees`.

RECOMMENDATION

Consider adding calls to `_revertIfRangeInvalid` to any functions that consume user-supplied position key data before that data is used, in the same way that `_claim` and `_withdraw` currently do. This solution is only viable as long as the `MAX_TICK_PRICE` fits within the space allotted for the `upper` and `lower` values in the `formatTokenId` function.

UPDATE

Fixed in pull requests #356 (commit hash `bfcaffa4a30ffc42db3fa552fa3d2b21042facea`) and #383 (commit hash `81209522480f99acf7d0903d70bdcda205e14044`). The Premia team added the relevant checks in `_revertIfRangeInvalid` to the `formatTokenId` function.

[N06] USE OF `TRANSFER` FUNCTION TO SEND ETHER

The `deployPool` function in the `PoolFactory` contract uses `transfer` to send ether to another address in the following locations:

- line 104
- line 107.

The use of the `transfer` function to send ether is no longer recommended. The `transfer` function forwards a fixed amount of gas (2300), and compatibility with receiving contracts may be broken if opcode gas costs are altered in the future.

RECOMMENDATION

Consider using the low-level `call` function to transfer ether to an address.

UPDATE

Fixed in pull request #371 (commit hash `2517a4e9758d2076bd4018af805fb5af04448659`), as recommended.

VULNERABILITY STATISTICS

Severity	Count
Critical	1
High	2
Medium	4
Low	12
Note	6

FIXES SUMMARY

Finding	Severity	Status
C01	Critical	Fixed in pull request #252
H01	High	Fixed in pull request #345
H02	High	Fixed in pull request #261
M01	Medium	Fixed in pull request #345
M02	Medium	Fixed in pull requests #386, #400, #402, and #411
M03	Medium	Fixed in pull request #260
M04	Medium	Fixed in pull request #345
L01	Low	Fixed in pull request #294
L02	Low	Acknowledged
L03	Low	Fixed in pull requests #372 and #382
L04	Low	Fixed in pull request #303
L05	Low	Fixed in pull request #353
L06	Low	Fixed in pull request #302
L07	Low	Fixed in pull request #361
L08	Low	Fixed in pull request #358
L09	Low	Fixed in pull request #345
L10	Low	Fixed in pull requests #229 and #350
L11	Low	Fixed in pull request #345
L12	Low	Acknowledged
N01	Note	Fixed in pull request #370
N02	Note	Fixed in pull request #384
N03	Note	Fixed in pull requests #364 and #379
N04	Note	Fixed in pull request #345
N05	Note	Fixed in pull requests #356 and #383
N06	Note	Fixed in pull request #371

APPENDIX

APPENDIX A: SEVERITY DEFINITIONS

Severity	**Definition**
Critical	This issue is straightforward to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
High	This issue is difficult to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
Medium	This issue is important to fix and puts a subset of users' data at risk and is possible to lead to moderate financial impact.
Low	This issue is not exploitable in a recurring basis and cannot have a significant impact on execution.
Note	This issue does not pose an immediate risk but is relevant to security best practices.

APPENDIX B: FILES IN SCOPE

```
contracts
└── adapter
│    └── FeedRegistry.sol
│    └── FeedRegistryStorage.sol
│    └── OracleAdapter.sol
│    └── Tokens.sol
│    └── chainlink
│    │    └── ChainlinkAdapter.sol
│    │    └── ChainlinkAdapterStorage.sol
│    └── composite
│    │    └── UniswapV3ChainlinkAdapter.sol
│    └── uniswap
│    │    └── UniswapV3Adapter.sol
│    │    └── UniswapV3AdapterProxy.sol
│    │    └── UniswapV3AdapterStorage.sol
└── factory
│    └── InitFeeCalculator.sol
│    └── PoolFactory.sol
│    └── PoolFactoryProxy.sol
│    └── PoolFactoryStorage.sol
└── libraries
│    └── ArrayUtils.sol
│    └── DoublyLinkedListUD60x18.sol
│    └── EnumerableSetUD60x18.sol
│    └── OptionMath.sol
│    └── Position.sol
│    └── PRBMathExtra.sol
│    └── Pricing.sol
└── oracle
│    └── VolatilityOracle.sol
│    └── VolatilityOracleStorage.sol
└── pool
│    └── PoolBase.sol
│    └── PoolCore.sol
│    └── PoolDepositWithdraw.sol
│    └── PoolInternal.sol
│    └── PoolProxy.sol
│    └── PoolStorage.sol
│    └── PoolTrade.sol
└── proxy
│    └── Premia.sol
│    └── ProxyUpgradeableOwnable.sol
│    └── ProxyUpgradeableOwnableStorage.sol
└── referral
│    └── Referral.sol
│    └── ReferralProxy.sol
│    └── ReferralStorage.sol
└── router
```

```
│   └── ERC20Router.sol
└── settings
│   └── UserSettings.sol
│   └── UserSettingsStorage.sol
└── vault
    └── VaultRegistry.sol
    └── VaultRegistryStorage.sol
    └── strategies
        └── underwriter
            └── UnderwriterVault.sol
            └── UnderwriterVaultProxy.sol
            └── UnderwriterVaultStorage.sol
```

APPENDIX C: CLIENT-IDENTIFIED BUG FIXES

During the audit engagement, the Premia team identified several bugs in contracts that fell within the scope of the audit. AE reviewed the following pull requests that contain fixes for those bugs:

- PR #326
  - Partially fixed as of commit hash `58ec7fc7628d6ff9817443de6694011a1ce7d8be`, with additional recommended changes made in pull request #391 (commit hash `62598b2b818325b6bd280c936a9b4f840f9aae41`) and pull request #409 (commit hash `e3361bc2cc1a5482513f1bbaca5f99b8cbff9920`). The Premia team acknowledged that the additional decimal places do not fully mitigate the effects of rounding, particularly in the case of tokens with low values, and is planning to make additional changes in the future to reduce the impact of rounding in those situations.
- PR #300
  - Fixed as of commit hash `89197acca89619127ca60b54cf282b5f83300e5c`
- PR #362
  - Fixed as of commit hash `937ef04189f008f257450a97d0f2bd064aa64adc`
- PR #231
  - Fixed as of commit hash `af0766773f1f4a03641c5c1b696deae0854061b8`
- PR #242
  - Fixed as of commit hash `d01d26aa717832aeb557863f5d025c95ae557312`, with additional recommended changes made in pull request #409 (commit hash `5dbd7bc61b1d2b63c00f9d5111566cea142ab7bd`)
- PR #248
  - Fixed as of commit hash `f6fd44c545aaa24a78f6bf67bf96dc1f60a1a8d7`
- PR #249
  - Fixed as of commit hash `17f97d3531a4dad98af0e5d9eaaf2b47c5ca07a8`
- PR #250
  - Fixed as of commit hash `407c5a20defc1bf1133d419e79278fa3d9f40485`
- PR #253
  - Fixed as of commit hash `3219c317b898cf39d8d9cbc79df0a445d1dc8bac`, with additional recommended changes made in pull request #401 (commit hash `be41dec0f0c949a8174a19a23293c4d564941ebf`)
- PR #254
  - Fixed as of commit hash `a910700bd6a13e9db557db04865cf6d26119e12a`
- PR #255
  - Fixed as of commit hash `8d9c29e0dba81c86b57371e37de7c3da6d285df7`
- PR #257
  - Fixed as of commit hash `3f3ee077859fe3337b399f4177e257a32738218b`
- PR #374
  - Fixed as of commit hash `321ac5108cec94b65a7111dfd54d1476c40616df`
- PR #225
  - Fixed as of commit hash `2ddb95b93c4f5693ef14e6ca362124cafbfb6abc`