December 2025

Smart Contract Audit Report

Client: **Radical**

Introduction

Scope

The audited files are located in the repository radical-finance/radical at **c16d826314d67b67311a2e78c520128d08fe7da4** commit.

The scope of the audit is limited to the following files:

- contracts/src/oracles/UniswapOracle.sol
- contracts/src/versionAdapter/UniswapTooling.sol
- contracts/src/libraries/TickConfigLib.sol
- contracts/src/strategies/UniswapV3Vault.sol

Summary

UniswapV3Vault is a yield-bearing vault that manages Uniswap V3 liquidity positions on behalf of depositors. The vault accepts deposits of two tokens, creates concentrated liquidity positions within a configured tick range, and allows users to earn trading fees generated by the underlying Uniswap V3 pool.

Users deposit token pairs into the vault and receive ERC20-compatible share tokens proportional to their contribution. These shares have special transfer behavior: the ERC20 **_update** function is overridden and enforces a cooldown system where incoming transfers reset the receiver's **lastDepositTimestamp**, blocking their withdrawals for **minDelayDepositSeconds**. Transfers to addresses still in their cooldown window (**minDelayDepositSeconds** + **TransferWindowSeconds**) are rejected unless the receiver has opted out via **setTransferCooldownOptOut**.

The vault maintains a single Uniswap V3 position represented by an NFT from the **NonfungiblePositionManager**. Trading fees accumulated by the position can be collected and reinvested to compound returns.

The vault is designed for stablecoin pairs where price remains close to parity. The tick range is configured as **TickBounds** (tickUpper/tickLower) at initialization. A TWAP oracle validates that the pool's spot price has not been manipulated beyond a configurable tick divergence threshold. All share pricing and value calculations in state-changing operations use the validated **spot price**, not the TWAP.

A configurable deposit-to-withdrawal delay prevents same-block deposit and withdraw, serving as the primary flash loan protection.

The withdrawal function has two overloads: **withdraw(balance)** validates the pool price against the oracle, while **withdraw(balance, amount0Min, amount1Min)** skips oracle validation and relies on the user-provided slippage parameters instead.

The contract uses upgradeable base contracts with an **initializer**-gated **initialize()** function, but is meant to be deployed directly without a proxy in this version. Future versions are intended to support upgradeability via a proxy pattern. All parameters set at initialization (tick range, oracle, fee tier, risk config) are permanent with no setters, making the vault fully immutable post-deployment.

The system consists of the following components:

- **UniswapV3Vault**: Core vault contract managing deposits, withdrawals, and liquidity operations.
- **UniswapOracle**: TWAP oracle for price queries from Uniswap V3 pools
- **UniswapTooling**: Adapter contract exposing Uniswap V3 library functions for Solidity 0.8 compatibility
- **TickConfigLib**: Library for tick range validation and construction.

Privileged Roles

Owner

The **owner** role is assigned during initialization. The owner can:

- **Initialize** the vault with configuration parameters including token pair, fee tier, tick range, and oracle settings
- Force withdraw funds on behalf of any user via the **returnFunds** function
- Standard ownership functions including transferring ownership

Fee Vault

The **feeVault** address receives external rewards claimed through the **claimMerkleFees** function. The fee vault can:

- Change itself to a new address via **setFeeVault**
- Receive all tokens claimed from Merkle distributor contracts

Trust Assumptions

Users depositing into the vault trust that:

- The owner will not maliciously force withdraw their funds
- The configured oracle provides accurate TWAP prices and is not compromised
- The Uniswap V3 pool and **NonfungiblePositionManager** contracts function correctly
- The tick range configuration is appropriate for the deposited token pair and will remain appropriate
- The **MaxTickDivergence** is set to a value that prioritizes security over user experience, effectively preventing price manipulation even if it occasionally blocks legitimate operations during volatile periods
- The fee vault is an EOA controlled by a trusted party that will distribute claimed rewards appropriately..

Issue Table

ID	Title	Severity	Status
C-01	Incorrect Price Conversion in **sqrtX96ToToken1PerToken0** Leads to Wrong Share Calculation	Critical	Resolved
H-01	Incorrect Position Value Calculation When Price Is Outside Range	High	Resolved
M-01	Vault Share Inflation Attack via Supply Reduction	Medium	Resolved
M-02	Missing Slippage Protection in Liquidity Operations	Medium	Resolved
M-03	Using TWAP for NAV Calculation Instead of Manipulation Check Creates Arbitrage Opportunity	Medium	Resolved
L-01	No Validation That Current Price Is Within Position Range	Low	Acknowledged, Not Resolved
L-02	Reentrancy Guard on Internal Functions May Cause Unexpected Reverts	Low	Resolved
I-01	Internal and External Functions with Identical Logic	Informational	Resolved
I-02	Unused Variable Fetched from slot0 in Oracle Contract	Informational	Resolved
I-03	No Validation of Tick Window Against Pool Tick Spacing	Informational	Resolved
I-04	Unused **tickerCenter** Configuration Parameter	Informational	Resolved
I-05	Potential Division by Zero in **sharesToUnderlyingLiquidity** When Supply is Zero	Informational	Resolved
I-06	Inconsistent Naming Convention for Internal Functions	Informational	Resolved

The final reviewed commit containing all fixes is :
21772819cf3c8ba6ea7fe1278b27bdc2036d8876

Severity Classification

- **Critical:** A practical, low-assumption exploit can cause **catastrophic impact** (e.g., large-scale theft/loss of assets, irreversible fund freeze, or full protocol takeover).

- **High:** An exploit can cause **severe impact** (e.g., significant asset loss, major privilege escalation, or major protocol disruption), but is **more constrained** (limited scope, additional conditions, or less-likely assumptions) than Critical.

- **Medium:** No direct, default-path asset theft, but the issue can **materially harm protocol function or value** (e.g., value leakage, partial DoS, incorrect accounting) under stated assumptions/constraints.

- **Low:** Assets are not meaningfully at risk, the issue is a **minor correctness/spec/edge-case** problem with limited impact or easy workarounds.

- **Informational:** No material security or correctness impact, improvements related to **style, clarity, documentation, events/monitoring, or best practices**.

Critical Severity Issues

***C-01**: Incorrect Price Conversion in **sqrtX96ToToken1PerToken0** Leads to Wrong Share Calculation*

The **sqrtX96ToToken1PerToken0** function is intended to convert an amount of token1 to its equivalent value in token0 using the Uniswap V3 **sqrtPriceX96** format. However, the mathematical formula implemented is incorrect; it multiplies by price instead of dividing by price.

In Uniswap V3, **sqrtPriceX96** represents **sqrt(token1/token0)** multiplied by **2^96**. To convert a token1 amount to its token0 equivalent, the formula should divide by price (token1/token0). The current implementation performs the inverse operation.

This bug directly affects the **issueShareToUsers** function, which uses **sqrtX96ToToken1PerToken0** to calculate the total value of a user's deposit in token0 terms. The incorrect conversion causes **totalValueOfDeposit** to be calculated incorrectly, which in turn affects the number of shares minted to depositors.

When price is greater than 1 (token0 is worth more than token1), depositors receive more shares than they should, effectively extracting value from existing vault depositors. When price is less than 1, depositors receive fewer shares than deserved.

The **_getTotalValueInToken0** function is also affected, causing the vault's total value to be misreported. This compounds the share calculation error and affects all value-based operations in the protocol.

Consider replacing the current formula with the correct price conversion.

High Severity Issues

***H-01**: Incorrect Position Value Calculation When Price Is Outside Range*

The vault incorrectly calculates the underlying token amounts of its Uniswap V3 position when the current price moves outside the position's tick range. Both **getUnderlyingUniswapPositionBalances** and **liquidityAmountToTokens** assume the current price is always within the position's range and do not handle the boundary cases required by Uniswap V3 concentrated-liquidity math.

In Uniswap V3, when the price moves outside a position's range, the position becomes entirely composed of one token. When the price is below the lower tick, all liquidity is held as token0. When the price is above the

upper tick, all liquidity is held as token1. The current implementation does not account for these cases.

Specifically, the functions call **getAmount0ForLiquidity** with the current price and the upper bound, and **getAmount1ForLiquidity** with the lower bound and the current price. When the current price is below the lower bound, **getAmount0ForLiquidity** receives an artificially wide range, producing an inflated token0 amount, while **getAmount1ForLiquidity** computes a positive value when the actual token1 amount should be zero. The inverse occurs when the price exceeds the upper bound.

The **getUnderlyingUniswapPositionBalances** function feeds into **_getTotalValueInToken0**, which is used to calculate shares minted to depositors. When price is outside the range, the total vault value is incorrectly calculated, causing depositors to receive an incorrect number of shares.

Depending on the direction and magnitude of the price deviation, depositors may receive more or fewer shares than their deposit warrants, creating unfair value transfer between users.

The **liquidityAmountToTokens** function is used by **positionBalances**, a view function that displays user positions. While this does not directly affect protocol state, users receive incorrect information about their holdings which could lead to poor financial decisions.

Consider replicating the logic from Uniswap's **getAmountsForLiquidity** function in the **LiquidityAmounts** library, which properly handles all three price scenarios.

Medium Severity Issues

***M-01**: Vault Share Inflation Attack via Supply Reduction*

The vault is susceptible to a share inflation attack that allows an attacker to steal deposits from subsequent users. While the attack requires real capital commitment due to the withdrawal time delay, it remains economically viable for sufficiently large victim deposits.

The first depositor receives a fixed **FIRST_SHARE** amount of 1e18 shares regardless of deposit size. After waiting for the required **minDelayDepositSeconds** period, the attacker can withdraw all but one share, reducing the total supply to 1. During this period, **no other deposits should occur.**

With the supply reduced to 1, the attacker monitors the mempool for incoming deposits. Upon detecting a victim's deposit transaction, the attacker front-runs it by donating tokens directly to the vault, inflating the value of the single remaining share.

When the victim's transaction executes, the share calculation results in zero shares due to integer truncation, the inflated vault value causes the division to round down to zero. The victim's tokens enter the vault but they receive no shares in return. The attacker then withdraws their single share, claiming both their donation and the victim's entire deposit.

Consider to implement a minimum supply that cannot be withdrawn. This ensures the supply can never be reduced below this threshold, making the required donation for an inflation attack economically prohibitive.

***M-02**: Missing Slippage Protection in Liquidity Operations*

The vault performs liquidity operations on Uniswap V3 without slippage protection. Both the **addLiquidity** function and the **withdraw** function set minimum output amounts to zero, providing no protection against price manipulation.

In **addLiquidity**, when minting a new position or increasing liquidity on an existing position, the parameters **amount0Min** and **amount1Min** are set to zero. This means the vault accepts any amount of liquidity regardless of how unfavorable the execution price is.

In the **withdraw** function, the **decreaseLiquidity** call also uses **amount0Min** and **amount1Min** of zero. This exposes withdrawing users to receiving significantly less tokens than expected.

An attacker can exploit this through sandwich attacks. When a user initiates a deposit that triggers **addLiquidity**, an attacker can front-run with a large swap to move the price unfavorably, let the victim's transaction execute at the bad price. The same attack applies to withdrawals. For deposits, users may receive less liquidity than expected for their tokens. For withdrawals, users may receive fewer tokens than their share of the pool warrants.

Consider implementing a slippage protection by adding minimum output parameters to the **depositExact** and **withdraw** functions. Users should specify their acceptable minimum amounts when calling these functions, allowing them to define their own slippage tolerance.

***M-03:** Using TWAP for NAV Calculation Instead of Manipulation Check Creates Arbitrage Opportunity*

The vault uses TWAP price from the oracle to calculate the total value of deposits and the vault's net asset value for share minting. This approach creates arbitrage opportunities because TWAP by design lags behind spot price.

When market prices move, there is a period where TWAP and spot price diverge. This divergence allows users to profit by depositing at times when TWAP misprices assets relative to the current spot price.

Additionally, the vault has an inconsistency where **getUnderlyingUniswapPositionBalances** uses slot0 to fetch the current spot price for calculating position token amounts, while the price conversion in **sqrtX96ToToken1PerToken0** uses TWAP. This mix of pricing sources compounds the issue.

The recommended pattern for vault implementations is to use spot price for value calculations while using TWAP as a safety check. If the difference between spot and TWAP exceeds a threshold, the transaction should revert as this indicates potential manipulation. This approach provides current accurate pricing while protecting against flash loan attacks.

Consider to use spot price for NAV calculations with TWAP as a manipulation check. Fetch the current price from slot0 for all value calculations. Before accepting deposits, compare the spot price against the TWAP oracle. If the deviation exceeds a configurable threshold, revert the transaction with an error indicating potential price manipulation.

Low Severity Issues

***L-01**: No Validation That Current Price Is Within Position Range*

The vault does not validate during initialization or deposits that the current pool price falls within the configured position range. The position is created with bounds at negative **tickerWindow** and positive **tickerWindow**, representing a range centered at tick 0 which corresponds to a price of 1.
If the vault is deployed when the pool price is outside this range, or if price moves outside the range after deployment, deposits become one-sided and the position earns no trading fees. While the contract comments acknowledge that fees are not collected when price is outside the range, there is no check to prevent deposits during such conditions.

Users may deposit into a vault where their capital will not earn fees due to price being outside the position range. For a stablecoin vault, significant deviation from the 1:1 price indicates a depeg event where accepting deposits may not be appropriate.

Consider adding a check during deposits that verifies the current pool tick is within the position range and revert otherwise.

L-02: *Reentrancy Guard on Internal Functions May Cause Unexpected Reverts*

The contract applies the **nonReentrant** modifier to the internal function **_investLiquidtokens**. This is an unusual pattern as reentrancy guards are typically applied to external or public functions that serve as entry points.

When **nonReentrant** is placed on an internal function, any external function that also has **nonReentrant** and calls this internal function will cause a revert due to the reentrancy lock already being held. This creates non-obvious dependencies between functions and can lead to unexpected reverts when composing function calls.

The current code functions correctly, but the unusual pattern increases maintenance risk. Future modifications that add reentrancy protection to external functions could inadvertently break internal call chains.

Consider moving the **nonReentrant** modifier to external entry point functions rather than internal functions. This follows the conventional pattern and makes the reentrancy protection boundaries clear. Internal functions should assume they are called within an already-protected context.

Informational Issues

I-01*: Internal and External Functions with Identical Logic*

The contract defines two functions that perform identical operations: **getOraclePricePerUnitToken0** as an external view function and **_getOraclePricePerUnitToken0** as an internal view function. Both functions take a token1 amount as input and return the result of calling **sqrtX96ToToken1PerToken0** with the oracle price.

This duplication is unnecessary, as the same logic can be exposed without maintaining two separate implementations.

Consider removing both functions and replace with a single public function. A public visibility allows the function to be called both externally by other contracts and users, and internally within the contract itself. Alternatively, the external function could simply call the internal function to avoid duplicating logic.

I-02*: Unused Variable Fetched from slot0 in Oracle Contract*

The **sqrtPriceX96** function in the **UniswapOracle** contract fetches the current tick from the pool's slot0 storage but never uses it. The function retrieves

multiple return values from slot0, including the tick, but only proceeds to calculate the TWAP price using the observe function. The fetched tick value is discarded, resulting in unnecessary gas consumption.

Consider removing the unused slot0 call entirely since the function only requires data from **pool.observe** to calculate the TWAP price.

***I-03**: No Validation of Tick Window Against Pool Tick Spacing*

The vault accepts a **tickerWindow** parameter during initialization without validating that it is a multiple of the pool's tick spacing. Uniswap V3 requires all position tick bounds to be multiples of the pool's tick spacing, which varies by fee tier.

If an administrator configures **tickerWindow** with a value that is not a multiple of the pool's tick spacing, the first call to addLiquidity will revert when attempting to mint the position. This would leave the vault in a non-functional state requiring redeployment.

Consider adding validation during initialization that **tickerWindow** is a multiple of the pool's tick spacing. The tick spacing can be retrieved from the pool contract.

***I-04:** Unused **tickerCenter** Configuration Parameter*

The contract declares and initializes a **tickerCenter** state variable that is never used. During initialization, **tickerCenter** is set from the configuration and an event is emitted, but the **addLiquidity** function creates positions centered at tick 0 using negative **tickerWindow** and positive **tickerWindow** as bounds, completely ignoring the **tickerCenter** value.

Consider to either remove the **tickerCenter** variable and related configuration if positions should always be centered at tick 0, or implement the intended functionality by using **tickerCenter** minus **tickerWindow** and **tickerCenter** plus **tickerWindow** as the position bounds.

***I-05:** Potential Division by Zero in **sharesToUnderlyingLiquidity** When Supply is Zero*

The **_sharesToUnderlyingLiquidity** function performs a division by supply without explicitly checking for the zero case. While the function returns early if balance is zero, it does not handle the edge case where supply could be zero.

The function relies on the invariant that if supply equals zero then all user balances must also be zero. However, explicitly validating this assumption provides defense in depth against potential state inconsistencies.

Consider adding an explicit check that reverts with a descriptive error message when supply is zero. This makes the precondition explicit and provides a clear error message if the invariant is ever violated.

I-06: *Inconsistent Naming Convention for Internal Functions*

The contract uses inconsistent naming conventions for internal functions. Some internal functions follow the convention of prefixing with an underscore such as **_withdraw**, **_collectFees**, and **_getTotalValueInToken0**, while others omit the underscore such as **addLiquidity**, **issueShareToUsers**, and **balanceOfBothTokens**.
Consistent naming conventions improve code readability and make it immediately clear whether a function is internal or external when reading the codebase.

Consider adopting a consistent naming convention where all internal functions are prefixed with an underscore.

Conclusion

This security audit focused on the UniswapV3Vault contract and its supporting oracle infrastructure. The vault implements a share-based accounting system for managing pooled Uniswap V3 liquidity with automated fee collection and reinvestment capabilities.

Key areas of concern identified during the review include price calculation accuracy, share minting logic, handling of edge cases such as out-of-range positions, and protection against common DeFi attack vectors including sandwich attacks and share inflation. The findings detailed in this report range from critical mathematical errors to informational code quality improvements