Pashov Audit Group

Napier Security Review

September 30th 2025 - October 23rd 2025

Contents

1. About Pashov Audit Group

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

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

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

2. Disclaimer

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

3. Risk Classification

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

Impact

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

Likelihood

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

4. About Napier

Napier V2 is a yield-stripping protocol that lets users earn fixed-rate yield on their deposits or trade future yield exposure through principal and yield tokens. It integrates with Uniswap V4 via custom hooks and rehypothecation to maximize LP capital efficiency and automate yield generation.

5. Executive Summary

A time-boxed security review of the **napierfi/napier-v2** repository was done by Pashov Audit Group, during which **Ch_301**, **Said**, **JCN**, **merlinboii** engaged to review **Napier**. A total of **18** issues were uncovered.

Protocol Summary

Project Name	Napier
Protocol Type	Interest Rate Derivatives
Timeline	September 30th 2025 - October 23rd 2025

Review commit hash:
- 7f1de6956883f976a95678ba749657b4c077caa7 (napierfi/napier-v2)

Fixes review commit hash:
- 3ac71f39eccfd4263ee1ae1942d09a99d10dfc6f (napierfi/napier-v2)

Scope

`Events.sol` `Factory.sol` `TokiHook.sol` `TokiHookLogic.sol` `ITokiHook.sol` `AssetPriceProvider.sol` `StateView.sol` `TokiLens.sol` `TokiQuoter.sol` `AccessManager.sol` `TokiPoolDeployer.sol` `PoolFeeModule.sol` `ChainlinkOracleFactory.sol` `TokiLinearChainlinkOracle.sol` `TokiTWAPChainlinkOracle.sol` `LinearPrice.sol` `TokiOracle.sol` `TWAPPrice.sol` `TokiPoolToken.sol` `Types.sol` `ApproximationParams.sol` `FeePctsPool.sol` `Flags.sol` `Packing.sol` `ContractValidation.sol` `CurrencySettler.sol` `FeePctsPoolLib.sol` `HookletLib.sol` `LibAccessGuard.sol` `LibApproximation.sol` `LibExpiry.sol` `LibOracle.sol` `LibPauseGuard.sol` `LibRehypothecation.sol` `LiquidityAmounts.sol` `TokiSwap.sol` `TokiSwapBinSearch.sol` `V4Rehypothecation.sol` `WrapperFactory.sol` `NapierV2Immutables.sol`

`PrincipalTokenRouter.sol` `SwapAggregatorRouter.sol` `TokiPoolRouter.sol`
`TransientState.sol` `V4Permit2Payments.sol` `VaultConnectorRouter.sol`
`Dispatcher.sol` `UniswapV4Router.sol` `FeeModule.so`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	17
Total findings	**18**

Summary of findings

ID	Title	Severity	Status
[M-01]	Updating oracle with post-swap `lnImpliedRate` allows TWAP manipulation	Medium	**Resolved**
[L-01]	`HookSwap` event shows wrong lp fee amount	Low	**Resolved**
[L-02]	Back-running can neutralize curator's `unwindVault()` action	Low	Acknowledged
[L-03]	`AssetPriceProvider` misses stale Chainlink price check	Low	Acknowledged
[L-04]	Single-sided liquidity calculation off with high `PrincipalToken` fees	Low	Acknowledged
[L-05]	Favorable swap surplus not returned to user in YT to Underlying conversion	Low	Acknowledged
[L-06]	Favorable swap outcomes not returned to user in Underlying-to-YT zap	Low	Acknowledged
[L-07]	`addLiquidity` refunds may return shares, potentially causing integration issues.	Low	**Resolved**
[L-08]	`initialAnchor` not validated to be `>= IWAD`	Low	**Resolved**
[L-09]	Missing event emission for privileged function	Low	**Resolved**
[L-10]	`TokiOracle` can benefit from further documentation	Low	**Resolved**
[L-11]	`v4-integration.md` docs should be updated	Low	**Resolved**
[L-12]	Struct with unused fields in `TokiQuoter`	Low	**Resolved**

ID	Title	Severity	Status
[L-13]	`TokiOracle` can potentially misprice LP tokens due to rehypothecation	Low	Acknowledged
[L-14]	Certain command sequences invoked via `UniswapV4Router` can be griefed	Low	Acknowledged
[L-15]	Missing read-only reentrancy protection in `TokiHook.getTotalBalances()`	Low	**Resolved**
[L-16]	`PrincipalToken` expiry and pause checks missing in quote functions	Low	**Resolved**
[L-17]	`salt` inefficiency in `quoteCreateAndAddLiquidity()` mismatches address	Low	**Resolved**

Medium findings

[M-01] Updating oracle with post-swap `lnImpliedRate` allows TWAP manipulation

Severity

Impact: Medium

Likelihood: Medium

Description

`TokiHookLogic.beforeSwap()` incorrectly updates the oracle observation with the new `lnImpliedRate`. As a result, the observation "closes" the entire elapsed interval using the **post-swap** rate, as if it were active for the whole period since the previous observation.

```
function beforeSwap(
    ...
) external returns (bytes4, BeforeSwapDelta, uint24) {
    PoolId id = key.toId();
    ITokiHook.PoolStorage memory state = $.s_states[id];

    --- SNIPPED ---
    // Compute swap result and update token account balances and fees in memory state
@1>  (BeforeSwapDelta returnDelta, SwapResult memory result) = _swap(env, key, params,
hookData, state, immutables);

    {
        // Write state to storage: any state updates must be blocked by reentrancy guard
        // Update oracle
@2>     _updateOracle($.s_observations[id], state);

        // Write state to storage ...
    }
    --- SNIPPED ---
}
```

The observation transformer multiplies the new rate by the entire time since the last observation.

```
function transform(Observation memory last, uint32 blockTimestamp, uint96 lnImpliedRate)
    internal
    pure
    returns (Observation memory)
{
    return Observation({
        blockTimestamp: blockTimestamp,
@3>     lnImpliedRateCumulative: last.lnImpliedRateCumulative
            + uint216(lnImpliedRate) * (blockTimestamp - last.blockTimestamp),
```

```
        initialized: true
    });
}
```

As the `state.lnImpliedRate` is updated during the swap (`@1>`), calling `_updateOracle` afterwards (`@2>`) causes the new observation to record the post-swap rate as if it had been active for the entire interval since the last observation (as shown in `@3>`).

This leads to inaccurate TWAP calculations, as the new rate is only active for a brief period at the end of the interval, not throughout the entire period. **As a result, a single swap can have a disproportionate impact on the TWAP, as the manipulated rate is treated as if it applied for the entire period.**

Recommendation

Update the oracle with the pre-swap `lnImpliedRate` , as using the pre-swap rate effectively records the previous block's closing price.

This aligns with Uniswap's in-block manipulation-resistance design: the first observation in a new block reflects the market price set by the last trade in the prior block. Any attempt to skew the oracle now requires the attacker to set the end-of-block price to influence the next block, which is costly.

Low findings

[L-01] `HookSwap` event shows wrong lp fee amount

The `HookSwap` event is emitted after every successful swap to provide off-chain indexers with data about the trade. The event includes a parameter named `hookLPfeeAmount0`, which is intended to represent the portion of the swap fee that is paid to the pool's Liquidity Providers (LPs).

However, the current implementation incorrectly passes the `result.swapFee` (the total swap fee) to this parameter. The `result.swapFee` includes both the share for the LPs and the share for the protocol/curator (`result.fees`).

As a result, the event emits the total fee under a name that implies it is only the LP portion, leading to misleading and incorrect data for any off-chain service that consumes this event.

Code Reference:

```
// in TokiHookLogic.sol#beforeSwap
(result.amount0, result.amount1, result.swapFee, result.fees) = TokiSwap.swap(...);

// ...

Events.emitHookSwap({
    // ...
    hookLPfeeAmount0: result.swapFee.toUint128(), // Incorrect: This is the total fee
    hookLPfeeAmount1: 0
});
```

Recommended Fix: Modify the event emission to correctly report only the portion of the fee that accrues to Liquidity Providers.

```
-         hookLPfeeAmount0: result.swapFee.toUint128(),
+         hookLPfeeAmount0: (result.swapFee - result.fees).toUint128(),
```

Or change the name if the `hookLPfeeAmount0` is misnamed.

[L-02] Back-running can neutralize curator's `unwindVault()` action

The `TokiHook` contract provides a permissioned `unwindVault()` function, allowing a curator to withdraw funds from an external rehypothecation vault in an emergency or for rebalancing purposes. However, the contract also contains a `_rebalance()` function that is automatically triggered after every swap to maintain the pool's `targetRawTokenRatio`.

A malicious user or MEV bot can exploit the interaction between these two functions. By observing a successful `unwindVault` transaction in the mempool, an attacker can immediately submit a back-run transaction that performs a small swap on the same pool within the same block.

The `unwindVault` action will move all assets from the vault to the hook's raw balance, pushing the raw token ratio to the `targetRawTokenRatio`. The subsequent swap from the back-runner will trigger the `_rebalance()` logic, which will detect that the raw ratio is above the configured maximum and automatically re-deposit a significant portion of the just-withdrawn assets right back into the same vault.

This effectively neutralizes the curator's administrative action, wastes the gas spent on the `unwindVault` call, and leaves the LPs' funds exposed to a potentially compromised vault that the curator was attempting to exit. Note: this is not possible if the PT has reached its maturity.

Recommendation The protocol should introduce a mechanism to allow the curator to temporarily suspend the automatic rebalancing feature.

[L-03] `AssetPriceProvider` misses stale Chainlink price check

The `tryGetPriceUSDInWad()` function in `AssetPriceProvider.sol` is responsible for fetching asset prices in USD from Chainlink price feed oracles. The function correctly calls `latestRoundData()` on the oracle contract to retrieve the price.

However, it fails to check the `updatedAt` timestamp that is returned alongside the price. There is no validation to ensure that the retrieved price data is recent.

If the Chainlink oracle network were to experience an outage or fail to update an on-chain price feed for an extended period, this function would continue to return the last known (stale) price as if it were current.

Recommended Fix:

```
// in AssetPriceProvider.sol
+   uint256 private constant STALE_PRICE_GRACE_PERIOD = 3 hours;

    // ... inside tryGetPriceUSDInWad ...
    try oracle.latestRoundData() returns (
        uint80, int256 retAnswer, uint256, uint256 updatedAt, uint80
    ) {
+       if (block.timestamp > updatedAt + STALE_PRICE_GRACE_PERIOD) {
+           // Price is stale, treat as if the call failed
+       } else {
            answer = retAnswer;
+       }
    } catch {}
```

[L-04] Single-sided liquidity calculation off with high `PrincipalToken` fees

The `_splitUnderlyingTokenLiquidityKeepYt` function calculates the optimal split of a single underlying token (YBT) for a liquidity provision. It calculates the `amount0ToTokenize` based on the current ratio of assets in the pool.

However, this calculation is performed under the incorrect assumption that the subsequent `principalToken.supply()` call is fee-less, (the issue is not that big if the `performanceFee` is reasonable). Because the calculation does not account for these fees, the amount of Principal Tokens (`PT`) that are actually minted is less than the amount expected by the split calculation. This results in an imbalanced pair of assets being passed to the final `TP_ADD_LIQUIDITY` command. The impact of this issue is proportional to the fee settings. While typically small, the `FeeModule` allows for performance and issuance fees as high as 100%, which could lead to significant and unavoidable value loss for users of this feature.

```
FeeModule.sol#initialize()
//..
        if (performanceFee > MAX_FEE_BPS) {
            revert Errors.FeeModule_PerformanceFeeExceedsMaximum();
        }
```

This can be avoided if the calculation in `_splitUnderlyingTokenLiquidityKeepYt()` is updated to account for the fees (only in case of big fees from the Principal Token) that will be deducted during the `supply` call. **Code Reference**:

```
TokiPoolRouter.sol#_splitUnderlyingTokenLiquidityKeepYt()

uint256 amount0ToTokenize = (amount0 * balances.value1()) / (balances.value1() +
TokiSwap.convertToAssets(balances.value0(), maxscale));
```

[L-05] Favorable swap surplus not returned to user in YT to Underlying conversion

The `YT_SWAP_YT_FOR_UNDERLYING` command facilitates a zap where a user sells YTs for the underlying token. The process involves combining the user's YTs with newly acquired PTs to redeem the underlying asset. The PTs are acquired via an internal exact-out swap (`underlying -> PT`), where the cost is estimated beforehand as `underlyingDebt` .

The final user payout is calculated as `underlyingWithdrawn - underlyingDebt` . However, `underlyingDebt` is a pre-calculated *maximum* input for the swap. The *actual* amount of the underlying token consumed by the swap can be less than `underlyingDebt` .

The function fails to account for this difference. It calculates the user's payout using the estimated maximum debt instead of the actual cost, causing any surplus funds from the swap to be permanently trapped in the `UniswapV4Router` contract.

It causes a small but direct loss of value for users selling YTs. The user does not receive the benefit of favorable market movement during the internal swap, and the resulting surplus is trapped in the router contract.

Code Reference:

```
TokiPoolRouter.sol#_swapYtForUnderlying()

uint256 underlyingDebt = uint256(-_underlyingDebt); // This is a pre-calculated maximum

// The actual swap happens inside combine(), and may consume less than underlyingDebt
uint256 underlyingWithdrawn = PrincipalToken(...).combine(...);

// This calculation uses the stale maximum debt, not the actual amount spent
uint256 amountOut = underlyingWithdrawn - underlyingDebt;

SafeTransferLib.safeTransfer(Currency.unwrap(key.currency0), recipient, amountOut); // This
transfers an incorrect, smaller amount
```

Recommended Fix:

```
-        SafeTransferLib.safeTransfer(Currency.unwrap(key.currency0), recipient, amountOut);
+        uint256 finalBalance = SafeTransferLib.balanceOf(Currency.unwrap(key.currency0),
address(this));
+        if (finalBalance > 0) {
+            SafeTransferLib.safeTransfer(Currency.unwrap(key.currency0), recipient,
finalBalance);
+        }
```

[L-06] Favorable swap outcomes not returned to user in Underlying-to-YT zap

The *YT_SWAP_UNDERLYING_FOR_YT* command in `Dispatcher.sol` facilitates a complex "zap" transaction where a user swaps an underlying token for a Yield Token (YT). This operation involves an internal swap where flash-minted Principal Tokens (PTs) are sold to acquire the underlying token debt. This internal swap is simulated to get an estimated input amount (bestUnderlying), but the actual swap execution can result in a more YBT amount.

The `onSupply()` callback function, which handles the settlement of this internal swap, correctly repays the debt to the PrincipalToken contract using the estimated `amountToTransfer` and `bestUnderlying` amount

```
File: src/zap/modules/TokiPoolRouter.sol#onSupply()

payOrPermit2Transfer( Currency.unwrap(key.currency0)/*token*/, payer,
Currency.unwrap(key.currency1)/*recipient PT*/, amountToTransfer.toUint160() );
 // ...
SafeTransferLib.safeTransfer(Currency.unwrap(key.currency0), msg.sender, bestUnderlying);
```

However, if the actual swap yields more of the underlying token than this estimated amount (bestUnderlying), the surplus (the "extra" funds from the swap) is not refunded to the user who initiated the transaction.

As a result, these surplus funds remain trapped in the `UniswapV4Router.sol` contract indefinitely. Code Reference:

```
 TokiPoolRouter.sol#onSupply()
function onSupply(uint256, /* underlyingDebt */ uint256 principals, bytes calldata data)
external override {
    // ...
    // Executes the swap, receiving the actual amount which may be > bestUnderlying
    _unlock(abi.encodeCall(poolManager.unlock, (abi.encode(COMMAND_V4_SWAP_EXACT_IN, inputs))));
    // ...
```

[L-07] `addLiquidity` refunds may return shares, potentially causing integration issues.

`addLiquidity` can return the leftover as ERC4626 vault shares to `refundReceiver` instead of the underlying tokens. Users/integrations may expect refunds in assets, not shares. If the receiver is a contract that doesn't handle vault shares, the refund could cause the funds to become stuck.

```
    function _deposit(DepositParams memory params) internal returns (DepositReturnData memory
returnData) {
        // ..

        // Take tokens that is going to lie in the pool (custodied by PoolManager) - this will
call unlockCallback.
        // Hook doesn't pull more tokens than the original amount user specified.\
        uint256 rawAmount0;
        {
           // ...

            if (refundShares0 > 0) {
>>>             SafeTransferLib.safeTransfer(address(params.vault0), params.refundReceiver,
refundShares0);
            }
        }

        uint256 rawAmount1;
        {
            // ...

            if (refundShares1 > 0) {
>>>             SafeTransferLib.safeTransfer(address(params.vault1), params.refundReceiver,
refundShares1);
            }
        }
```

If keeping current behavior, clearly document it and emit explicit events indicating shares were refunded.

[L-08] `initialAnchor` not validated to be `>= IWAD`

The `initialAnchor` is used to adjust the interest rate around a point at which trading will be the most capital efficient. If `initialAnchor < IWAD`, it implies a negative base rate, which can result in the initial `lnImpliedRate` (and by extension, the initial `exchangeRate`) being computed as a value `< IWAD` when adding initial liquidity. Therefore, an `initialAnchor < IWAD`, can cause the initial liquidity provision for a pool to revert:

```
    function _getExchangeRateNoFee(
        uint256 totalAssets,
        uint256 totalPt,
        int256 rateScalar,
        int256 rateAnchor,
        int256 netPtToAccount
    ) internal pure returns (int256 exchangeRate) {
        int256 newTotalPt = totalPt.toInt256() - netPtToAccount;

        if (newTotalPt < 0) {
            revert Errors.TokiSwap_InsufficientPrincipalsLiquidity();
        }

        int256 proportion = newTotalPt * IWAD / (totalPt + totalAssets).toInt256();

        // Sanity check - Too much principal token is going to be in the pool
        if (proportion > MAX_PROPORTION) {
            revert Errors.TokiSwap_MarketProportionTooHigh();
        }

        int256 lnProportion = _lnProportion(proportion);
        exchangeRate = lnProportion * IWAD / rateScalar + rateAnchor;

        // Sanity check - Negative implied rate / Principal token in in premium is not allowed
        if (exchangeRate < IWAD) revert Errors.TokiSwap_ExchangeRateBelowOne(exchangeRate);
    }
```

Ideally, users will compute optimal `scalarRoot` and `initialAnchor` values via the `TokiQuoter::computePoolParameters` function, which should not yield an `initialAnchor < IWAD`. However, it's possible that users can compute these values via different means, and therefore it is best practice to validate both the `scalarRoot` and `initialAnchor` during pool deployment.

However, currently only the `scalarRoot` is validated:

```
            (uint256 scalarRoot, int256 initialAnchor) = abi.decode(ammParams, (uint256,
int256));
            if (scalarRoot == 0) Errors.TokiHook_InvalidScalarRoot.selector.revertWith();

            requestedCardinalityNext = cardinalityNext;
            immutables.scalarRoot = scalarRoot;
            immutables.initialAnchor = initialAnchor;
```

For comparison, Pendle requires that the supplied `initialAnchor` is `>= 1e18` upon market creation:

```
    int256 public constant minInitialAnchor = PMath.IONE;
...
    function createNewMarket(
        address PT,
        int256 scalarRoot,
        int256 initialAnchor,
        uint80 lnFeeRateRoot
    ) external returns (address market) {
...
        if (initialAnchor < minInitialAnchor)
            revert Errors.MarketFactoryInitialAnchorTooLow(initialAnchor, minInitialAnchor);
```

Consider validating `initialAnchor` upon pool deployment as well.

[L-09] Missing event emission for privileged function

The `FeeModule::updateFeeSplitRatio` function updates the critical state, but is missing an event emission:

```
    function updateFeeSplitRatio(uint256 _splitRatio) external
restrictedBy(i_factory().i_accessManager()) {
        if (_splitRatio > MAX_SPLIT_RATIO_BPS) {
            revert Errors.FeeModule_SplitFeeExceedsMaximum();
        }
        if (_splitRatio == 0) {
            revert Errors.FeeModule_SplitFeeTooLow();
        }
        s_feePcts = FeePctsLib.updateSplitFeePct(s_feePcts, _splitRatio.toUint16());
    }
```

For comparison, the `PoolFeeModule::updateFeeSplitRatio` function includes an event emission to record the state change.

Consider emitting an event in `FeeModule::updateFeeSplitRatio`.

[L-10] `TokiOracle` can benefit from further documentation

Currently, `TokiOracle.md` suggests that users can specify a `twapWindow` of `0` to acquire the spot price from the last stored `lnImpliedRate`. However, in Toki Pools the `lnImpliedRate` changes continuously due to rehypothecation vault yield. This means that the exchange rate is continuously changing, and the last recorded `lnImpliedRate` in the pool's state will likely not equate to the actual spot `lnImpliedRate` for the pool.

Therefore, it should be explicitly documented that the `TokiOracle` should *not* be used to compute the spot price of assets. It should be noted that the spot price should exclusively be read from `TokiQuoter`, which takes into account an updated `lnImpliedRate`:

```
// TokiQuoter.sol

        // CHANGED: Rehypothecation Awareness
        // Dry-run swap to refresh lnImpliedRate due to rehypothecation vault interest
```

```
        _dryRunSwap(poolState, immutables, feePcts, true, 0);
...
    function _dryRunSwap(
        TokiSwap.PoolState memory poolState,
        ITokiHook.ImmutableParams memory immutables,
        FeePctsPool feePcts,
        bool zeroForOne,
        int256 amountSpecified
    ) internal view returns (int256 underlyingAmount, int256 principals) {
        ApproximationParams memory zeroApprox;
        // CHANGED: Dry-run Swap for Rate Refresh
        // Dry-run swap with 0-amount to refresh the lnImpliedRate without executing trades.
        //
        // KEY DIFFERENCE from Pendle:
        // - Pendle: lnImpliedRate remains static between trades
        // - Napier: lnImpliedRate changes continuously due to rehypothecation vault yield
accrual
        //
        // Why this is needed:
        // - Rehypothecation vaults (e.g., Aave, Compound) earn interest continuously
        // - This changes the underlying asset value and affects PT pricing
        // - Without this refresh, quotes would become stale and inaccurate over time
        // - The dry-run simulates market dynamics to get current fair value
```

Another consequence of the `lnImpliedRate` continuously changing is that the TWAP prices for assets will naturally deviate further from the spot price as time passes. This occurs even when the pool is idle, i.e. no swapping takes place.

Additionally, the TWAP price can deviate considerably from the spot price if one of the configured rehypothecation vaults experiences a sudden spike in yield, which may occur naturally (although unlikely) or can occur if the vault's share price is manipulated. In this case, the TWAP price can be drastically under or over-priced compared to the spot price. Bad actors may be able to take advantage of this mispricing when the asset is used in integrated protocols, i.e. as, collateral in a lending protocol.

[L-11] `v4-integration.md` docs should be updated

In `v4-intergration.md`, the Example Workflow section, `Create new pool and add liquidity spending wstETH as initial liquidity` example, does not specify the correct input values for the `TP_ADD_LIQUIDITY` command:

```
const inputs = [
  // 1. PERMIT2_PERMIT_BATCH
  {
    data: viem.encodeAbiParameters(
      ABI_DEFINITION[CommandType.PERMIT2_PERMIT_BATCH],
      permitBatchParams
    ),
  },

  // 2. TP_CREATE_POOL
  {
    data: viem.encodeAbiParameters(
```

```
      ABI_DEFINITION[CommandType.TP_CREATE_POOL],
      createPoolParams
    ),
  },
  // 3. TP_SPLIT_INITIAL_LIQUIDITY
  {
    data: viem.encodeAbiParameters(
      ABI_DEFINITION[CommandType.TP_SPLIT_INITIAL_LIQUIDITY],
      [ZERO_KEY, amount0, MSG_SENDER, desiredImpliedRate]
    ),
  },

  // 4. TP_ADD_LIQUIDITY
  // Pass ZERO_KEY to load pool key right after creating the pool
  {
    data: viem.encodeAbiParameters(
      ABI_DEFINITION[CommandType.TP_ADD_LIQUIDITY],
      [ZERO_KEY, amount0, amount1, liquidityMinimum, ADDRESS_THIS]
    ),
  },
];
```

When supplying initial liquidity, the `TP_SPLIT_INITIAL_LIQUIDITY` command will result in the YBT and PT amounts to add being stored in the router's balance.

Therefore, the subsequent `TP_ADD_LIQUIDITY` command should specify the `ActionConstants.CONTRACT_BALANCE` flag as the PT and YBT amounts instead of `amount0` and `amount1`.

[L-12] Struct with unused fields in `TokiQuoter`

The `QuoteRemoveLiquidityResult` struct in `TokiQuoter` is solely used as the return value for `TokiQuoter::quoteRemoveLiquidity`. However, only two fields of this struct are actually populated in this function:

```
    struct QuoteRemoveLiquidityResult {
        uint256 amount0Out;
        uint256 amount1Out;
        int256 spotExchangeRateBefore;
        int256 executionExchangeRate;
        int256 priceImpact;
    }

    function quoteRemoveLiquidity(PoolKey memory key, uint256 liquidity)
        public
        view
        checkPoolKey(key)
        returns (QuoteRemoveLiquidityResult memory result)
    {
..
        // Amounts withdrawn from vaults + raw amounts
        result.amount0Out = assets0 + rawAmount0;
        result.amount1Out = assets1 + rawAmount1;
```

As we can see above, the 3 fields pertaining to price impact context are not utilized. Additionally, the documentation in `TokiQuoter.md` assumes that `quoteRemoveLiquidity` will return price impact context, which it does not.

Consider updating the `QuoteRemoveLiquidityResult` struct to only contain the `amount0Out` and `amount1Out` fields, and update documentation to reflect this change.

[L-13] `TokiOracle` can potentially misprice LP tokens due to rehypothecation

Napier adopts Pendle's approach for pricing LP tokens, which models a hypothetical trade that moves the current spot implied rate toward the TWAP implied rate. The accuracy of this model is dependent on Pendle's assumption that pool proportions remain constant between trades. However, this assumption does not hold for Napier since, in Toki Pools, the proportion of the pool can continuously change due to rehypothecation vault yields. These proportional changes directly affect the implied rate (`lnImpliedRate`), meaning the spot implied rate is continuously changing as time passes.

However, the current implementation of the LP oracle, implemented in `TWAPPrice::_convertLpToAssetsRaw`, computes the `rateBlended` as if the `lnImpliedRate` remains static between trades:

```
(int256 rateOracle, int256 rateBlended) =
    _computeExchangeRates(poolState.lnImpliedRate, key, twapWindow, expiry); //
`poolState.lnImpliedRate` is stale previously stored rate
```

```
    function _computeExchangeRates(uint96 lastLnImpliedRate, PoolKey memory key, uint32
twapWindow, uint256 expiry)
        private
        view
        returns (int256 rateOracle, int256 rateBlended)
    {
        uint256 timeToExpiry = expiry - block.timestamp;

        uint256 twapLnImpliedRate = getTwapLnImpliedRate(key, twapWindow);
        rateOracle = TokiSwap.convertToExchangeRate(twapLnImpliedRate, timeToExpiry);

        int256 rateLastTrade = TokiSwap.convertToExchangeRate(lastLnImpliedRate,
timeToExpiry); // @audit: exchange rate does not reflect rehypothecation yield

        rateBlended = (rateLastTrade + rateOracle) / 2;
    }
```

The function `_computeExchangeRates` uses the stale `poolState.lnImpliedRate` (from current stored `lnImpliedRate`) to compute `rateLastTrade`, while other parameters (i.e., `ammParams.rateAnchor`, `ammParams.totalAssets`, `balances.value1()` and `balances.value0()`) are derived from the updated pool balances, which include vault yield.

This introduces an asymmetry in the overall equation, as all other parameters reflect updated balances that include vault yield, while `rateLastTrade`, and ultimately `rateBlended`, is computed using an outdated implied rate that does not correspond to those updated balances.

As a result, the LP token price computed by the oracle can deviate further than intended from the spot price. The magnitude of this mispricing is expected to be small under normal circumstances (high trading volume and modest, consistent vault yield), but could potentially be larger for low activity pools or when vault yields accrue significantly between swaps.

Consider performing a null "dry run swap" (similar to how it is done in `TokiQuoter`) in order to refresh the `poolState.lnImpliedRate` to ensure the `rateBlended` is consistent with current pool proportions.

[L-14] Certain command sequences invoked via `UniswapV4Router` can be griefed

The `ActionConstants.CONTRACT_BALANCE` flag is used to indicate that the router should use its own internal balance for the specified token amount. This implies that the user has previously transferred the necessary assets to the router prior to the command that utilizes this flag for a token amount, or the router has received token amounts on behalf of the user from a previous command in the sequence. This provides flexibility for complex command sequences, however, certain command sequences can be griefed by external actors donating tokens to the router prior to a user's transaction, which ultimately manipulates the token amounts utilized by the router for subsequent commands.

Below are three examples in which this can occur:

1. A user wishes to split their YBT into YBT + PT amounts and then add these amounts as initial liquidity into a Toki Pool. The user prepares a command sequence of `PERMIT2_PERMIT_BATCH` + `TP_SPLIT_INITIAL_LIQUIDITY` + `TP_ADD_LIQUIDITY`. The `PERMIT2_PERMIT_BATCH` command will permit the router to handle the user's specified `amount0` of YBT for the subsequent `TP_SPLIT_INITIAL_LIQUIDITY` command. The `TP_SPLIT_INITIAL_LIQUIDITY` command will then use a portion of the supplied YBT to mint PT and YT to the router. The PT will remain in the router while the YT gets sent to the user. The remaining YBT will also remain in the router. The remaining YBT and PT received will correspond to a pool proportion that will yield an initial `lnImpliedRate` equal to the user's specified `desiredImpliedRate`.

```
    /// @notice Split part of initial liquidity into PTs and spending them issuing PTs and YTs
    /// @dev This function means to be used with `TP_ADD_LIQUIDITY`
    /// @param receiver YT receiver address. PTs are issued to router contract
    /// @param desiredImpliedRate The desired implied rate in wad (Note e.g. 0.185e18 for 18.5%)
    function _splitInitialLiquidity(PoolKey memory key, uint256 amount0, address receiver,
uint256 desiredImpliedRate)
        internal
    {
...
        // Transfer underlying token to the contract
```

```
        amount0 = payIfNeeded(Currency.unwrap(key.currency0), amount0);

        approveIfNeeded(Currency.unwrap(key.currency0), address(pt));

        // Issue PTs and YTs with part of underlying token liquidity
        // Remaining amount of underlying token `amount0 - amount0ToTokenize` is going to be
left in the contract, then deposited to the pool later
        uint256 amount0ToTokenize = (amount0 * initialProportion) / 1e18;
        uint256 principals = pt.supply(amount0ToTokenize, address(this));

        // Send YTs to receiver
        SafeTransferLib.safeTransfer(address(pt.i_yt()), receiver, principals);
    }
```

As we can see above, the `TP_SPLIT_INITIAL_LIQUIDITY` command is meant to be used in combination with `TP_ADD_LIQUIDITY` and therefore the input data for the `TP_ADD_LIQUIDITY` command is required to specify the `CONTRACT_BALANCE` flag as the amount of YBT and PT to add (already held in router's balance from the previous command).

Therefore, it is possible for an external actor to manipulate the YBT amount that the router will utilize for the `TP_ADD_LIQUIDITY` command by donating YBT to the router ahead of the user's transaction. This will result in the user's `TP_ADD_LIQUIDITY` adding an inflated YBT amount as initial liquidity. Since the initial `lnImpliedRate` is computed in proportion to the pool's PT proportion, this inflated YBT amount will result in a deflated PT proportion, which ultimately results in a deflated initial `lnImpliedRate` with respect to the `desiredImpliedRate` originally specified by the user.

1. **Note that this particular sequence may be unlikely to occur in practice, as a user would likely opt to use PERMIT2 instead for token transfers. However, it is possible that another complex sequence can result in the router receiving PT and YT on behalf of the user from previous commands and then using those received amounts for the `PT_COMBINE` command. The sequence in this example is used to showcase how this particular `PT_COMBINE` command is susceptible to griefing.**

   A user wishes to combine their PT and YT into YBT. The user prepares a command sequence of `TRANSFER` + `TRANSFER` + `PT_COMBINE`. The PT and YT are first transferred to the router (or the router can receive these amounts on behalf of the user as part of another complex sequence), and then the `PT_COMBINE` command will specify the `CONTRACT_BALANCE` flag as the `principals` amount of PT and YT to combine (router will use its own balance).

```
    function _principalTokenCombine(PrincipalToken principalToken, uint256 principals, address
receiver) internal {
        ContractValidation.checkPrincipalToken(_i_factory, address(principalToken));

        address yt = address(principalToken.i_yt());
        uint256 actualPrincipals = payIfNeeded(address(principalToken), principals);
        uint256 actualYts = payIfNeeded(yt, principals);

        if (actualYts < actualPrincipals)
Errors.Zap_InsufficientYieldTokenBalance.selector.revertWith();
        principalToken.combine(actualPrincipals, receiver);
    }
```

As we can see above, the amount of YT in the router in this case must be `>=` the amount of PT, else the transaction reverts.

This check allows an external actor to grief the user's transaction by sending at least `1 wei` of PT to the router contract ahead of the user's transaction. This will result in the `actualPrincipals` being greater than the `actualYTs` held in the router's balance, and thus the user's transaction will revert.

1. **Note that this scenario is the least likely to occur, however, it can potentially lead to a loss of user assets.**

   A user wishes to combine their PT and YT into YBT. Suppose the router already has a PT balance of `1`. The user attempts to take advantage of this by preparing the same command sequence seen in example `2`. This time the user transfers `9` PT and `10` YT to the contract, anticipating that the total balance of the router during the `PT_COMBINE` command will be `10` PT and `10` YT.

   However, an external actor front runs the user's transaction and uses the `SWEEP` command to extract the `1` PT from the router before the user's transaction executes. As a result, the `actualYts` (equal to `10`) will be greater than the `actualPrincipals` (equal to `9`). Notice that `_principalTokenCombine` does not require the `actualPrincipals` and `actualYTs` amounts to be the same and it uses the `actualPrincipals` amount for the `PrincipalToken::combine` call. This means that only `9` of the user's PT and YT will be used to combine. The external actor can then back run the user's transaction with another `SWEEP` command to extract the user's remaining `1` YT.

   Recommendations Consider documenting these edge cases and noting that `SWEEP` commands can be used at the top, or bottom of certain command sequences in order to prevent external actors from potentially manipulating tokens amounts for subsequent commands.

[L-15] Missing read-only reentrancy protection in `TokiHook.getTotalBalances()`

The `TokiHook.getTotalBalances()` is widely used in `TokiQuoter` `StateView`, `TWAPPrice`, `TokiPoolRouter`, and `TokiSwapBinSearch`, but may return inconsistent state when called during `addLiquidity()` or `removeLiquidity()` via hooklet transfer callbacks or vault callbacks, potentially leading to incorrect state views or incorrect price calculations.

```
function getTotalBalances(PoolId id) external view returns (Uint128x2) {
    PoolStorage storage $ = s_hookStorage.s_states[id];
    address pointer = $.immutableParamsPointer;

    ContractValidation.checkTokiPoolExists(pointer);

    (address vault0, address vault1) = ImmutableParamsLib.getVaults(pointer);
    return LibRehypothecation.getTotalBalances({
```

```
        vault0: ERC4626(vault0),
        vault1: ERC4626(vault1),
        reserves: $.reserves,
        rawBalances: $.rawBalances
    });
}
```

Recommendation

Consider adding read-only reentrancy protection to `getTotalBalances()` or explicitly document the potential vulnerability.

Additionally, restructure liquidity operations to perform LP token minting/burning after state changes are finalized, as this action can trigger the hooklet transfer callbacks.

[L-16] `PrincipalToken` expiry and pause checks missing in quote functions

Several quote functions in `TokiQuoter` rely on `PrincipalToken.previewSupply()` and `PrincipalToken.previewIssue()` to calculate expected outputs.

However, these preview functions return `0` when the PrincipalToken is **paused or expired**, causing quotation functions to either return misleading results or implicitly revert.

Additionally, there are no explicit checks once the PrincipalToken expires for adding liquidity related functions, as once expired, the PT is not available for adding liquidity or swapping.

Recommendation

Add explicit checks for paused or expired states in quotation functions and revert with descriptive error messages when operations are not available.

[L-17] `salt` inefficiency in `quoteCreateAndAddLiquidity()` mismatches address

`TokiQuoter.quoteCreateAndAddLiquidity()` mines a `salt` to ensure the PT address is greater than the `currency0` address for proper Uniswap V4 pool creation. However, the selected salt is inefficient as it will ultimately be rehashed in the factory.

As a result, the factory will use a different `salt` than the one predicted by the quoter, leading to an incorrect address prediction, and can cause the execution to revert.

```
function quoteCreateAndAddLiquidity(
    ...
    address currency0
) external returns (PreviewAddLiquidityResult memory) {
    uint256 salt;
    Factory factory = principalTokenQuoter().factory();
    while (true) {
        assembly {
```

```
            let m := mload(0x40)
            mstore(m, chainid())
            mstore(add(m, 0x20), address())
            mstore(add(m, 0x40), salt)
@>          salt := keccak256(m, 0x60)
        }

@>       address predictedAddress =
            LibBlueprint.computeCreate2Address(bytes32(salt), suite.ptBlueprint,
address(factory));
        if (predictedAddress > currency0) break;
        unchecked {
            salt++;
        }
    }

@>   (,, address pool) = factory.deployDeterministic(suite, modules, expiry, msg.sender,
bytes32(salt));
    --- SNIPPED ---
}
```

However, `Factory._deploy()` rehashes the `salt` again before using it:

```
function _deploy(...)
    internal
    returns (address pt, address yt, address pool)
{
    --- SNIPPED ---
    // Switch salt calculation for true deterministic deployment
    // If the salt is not provided, use the default way.
    if (salt == bytes32(0)) {
        --- SNIPPED ---
    } else {
        // Re-hash the user-provided salt to avoid using the same salt of the above salt.
@>      salt = EfficientHashLib.hash(block.chainid, uint256(uint160(msg.sender)),
uint256(salt));
    }
    --- SNIPPED ---
}
```

This creates the sequence of salt that: 1. In quoter: `salt_A: hash(chainid, address(this), 0)` for `predictedAddress` and checks if `predictedAddress > currency0`. 2. If `salt_A` is valid, return `salt_A`, otherwise, `salt_A++` and repeat step 1. 3. In factory: `salt_B: hash(chainid, uint256(uint160(msg.sender)), uint256(salt_A))` for `pt` deployment. 4. The actual PT address is computed using `salt_B`, which is different from `salt_A`. 5. As a result, **if the actual PT address is not greater than `currency0`, the execution will revert even if we have predicted the `salt` upfront.**

Recommendation

Update `TokiQuoter.quoteCreateAndAddLiquidity()` to use the original `salt` (before hashing) that is used to predict the PT address in the quoter.

Additionally, avoid using or starting `salt` with `bytes32(0)` in the quoter, as this will also cause the factory to use a different salt calculation, leading to incorrect predictions.