Pashov Audit Group

Covenant Security Review

August 18th 2025 - August 31st 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 Covenant

Covenant is a platform for transforming risk by splitting a base-asset into two fungible, tradeable components: yield-coins, which accrue interest and provide safer exposure, and leverage-coins, which carry amplified risk and reward. Through perpetual debt mechanics and a latent swap invariant, Covenant markets operate as capital-efficient, fully collateralized lending exchanges that self-stabilize and enable dynamic risk and yield transformation.

5. Executive Summary

A time-boxed security review of the **covenant-labs/covenant-core** repository was done by Pashov Audit Group, during which **ast3ros**, **Tejas Warambhe**, **Said**, **Ch_301** engaged to review **Covenant**. A total of **25** issues were uncovered.

Protocol Summary	
Project Name	Covenant
Protocol Type	Interest Rate Derivatives
Timeline	August 18th 2025 - August 31st 2025

Review commit hash:
- 2075992fd533af9da64fbdaf690d0b8627110261 (covenant-labs/covenant-core)

Fixes review commit hash:
- a4521cde950b712db14e7ed550af858fd1c73bd0 (covenant-labs/covenant-core)

Scope

`CovenantLiquid.sol` `LatentSwapLEX.sol` `FixedPoint96.sol` `LatentMath.sol` `SqrtPriceMath.sol` `DataTypes.sol` `Errors.sol` `NoDelegateCall.sol` `Utils.sol` `ExchangeLogic.sol` `StateLogic.sol` `TransferLogic.sol` `ValidationLogic.sol` `DebtMath.sol` `PercentageMath.sol` `WadRayMath.sol` `MetaProxyDeployer.sol` `ERC20.sol` `SynthToken.sol` `interfaces/`

6. Findings

Findings count

Severity	Amount
High	1
Medium	5
Low	19
Total findings	**25**

Summary of findings

ID	Title	Severity	Status
[H-01]	Not excluding `accruedProtocolFee` from state update operations causes several issues	High	Resolved
[M-01]	`SynthToken` returns incorrect name, symbol, and decimals	Medium	Resolved
[M-02]	Market DoS when `baseTokenSupply` is 0 while Synth token supply is non-zero	Medium	Resolved
[M-03]	ETWAP logic allows grief attacks on redeem flow	Medium	Resolved
[M-04]	Redeem cap bypassed using `swap()`	Medium	Resolved
[M-05]	Early market can be DoSed by over-minting debt	Medium	Resolved
[L-01]	Protocol can be DOS due to arithmetic underflow in interest accrual	Low	Resolved
[L-02]	Inconsistent price reset behavior in full redemption scenarios	Low	Resolved
[L-03]	Missing zero amount validation for non-base asset outputs in swaps	Low	Resolved
[L-04]	Missing validation for identical base and quote Tokens in market	Low	Acknowledged
[L-05]	Potential division-by-zero in `_dexToSynth`	Low	Resolved
[L-06]	ETWAP check blocks the last user from immediate full withdrawal	Low	Resolved
[L-07]	Preview functions fail to verify the outputs	Low	Resolved

ID	Title	Severity	Status
[L-08]	Incorrect concat usage corrupts token name and symbol	Low	Resolved
[L-09]	Market state broken: `mint()` disabled but swaps creating debt allowed	Low	Resolved
[L-10]	Inconsistent LTV blocks zToken redemptions during undercollateralization	Low	Resolved
[L-11]	Inconsistent redeem cap allows excessive withdrawals	Low	Resolved
[L-12]	Incorrect 512-bit number comparison in computeLiquidity function	Low	Resolved
[L-13]	Protocol fee updates cause loss of accrued fees	Low	Resolved
[L-14]	`redeem` caps inputs for calculation but burns uncapped amounts, causing reverts	Low	Resolved
[L-15]	`swap` operation can lead to an undercollateralized state	Low	Resolved
[L-16]	Loss of protocol fee due to lack of decimal consideration during accrual	Low	Resolved
[L-17]	Market pause does not stop interest and fee accrual	Low	Acknowledged
[L-18]	The last withdrawer can be sandwiched to extract fees from the `redeem` operation	Low	Acknowledged
[L-19]	Approximate swap fee used for exact_out instead of exact calculation	Low	Resolved

High findings

[H-01] Not excluding `accruedProtocolFee` from state update operations causes several issues

Severity

Impact: Medium

Likelihood: High

Description

Covenant governance can turn on protocol fees, which if done, can cause several issues.

When `_calculateMarketState` is called, it calculates the accrued fee based on `baseTokenSupply`. This fee is added to `protocolFeeGrowth` and deducted from `baseTokenSupply`. However, several operations after the fee accrual still use the `baseTokenSupply` value that includes the fee, which causes several issues.

- The `lastETWAPBaseSupply` update will use an incorrect `baseTokenSupply`, since the accrued fee decreases `baseTokenSupply`. This results in an overestimation of the actual `baseTokenSupply` that should be used to update `lastETWAPBaseSupply`.

```
// ...
            // If baseTokenSupply has decreased since the last update, then decrease the
ETWAPBaseSupply tracker
            if (marketState.baseTokenSupply < marketState.lexState.lastETWAPBaseSupply)
                marketState.lexState.lastETWAPBaseSupply =
                    marketState.lexState.lastETWAPBaseSupply.rayMul(updateFactor) +
                    marketState.baseTokenSupply.rayMul(WadRayMath.RAY - updateFactor); // eTWAP
update if baseSupply decreased vs last update
        }

        // if baseTokenSupply has increased (even in same timeblock), update record
        if (marketState.baseTokenSupply >= marketState.lexState.lastETWAPBaseSupply)
            marketState.lexState.lastETWAPBaseSupply = marketState.baseTokenSupply; //
immediatedly update if baseSupply increased vs last update
```

- The `baseTokenSupply` converted to `liquidity` is also used to determine `maxDebtValue`. This overestimates `maxDebtValue`, which is very dangerous as it could incorrectly indicate that the market is not undercollateralized.

```
// ...
            marketState.liquidity = _synthToDex(marketState, baseTokenSupply, AssetType.BASE,
Math.Rounding.Floor)
                .toUint160();

            // Calculate debt balanced value from zTokenSupply
```

```
            marketState.dexAmounts[0] = _synthToDex(
                marketState,
                marketState.supplyAmounts[0],
                AssetType.DEBT,
                Math.Rounding.Floor
            );

            // Check max value for debt, given availablie liquidity in market.
            uint256 maxDebtValue = SqrtPriceMath.getAmount0Delta(
                _edgeSqrtPriceX96_A,
                _edgeSqrtPriceX96_B,
                marketState.liquidity,
                Math.Rounding.Floor
            );
// ...
```

- Which also used to calculate `dexAmounts[1]` and `lastSqrtPriceX96`.

```
// ...
            (marketState.dexAmounts[1], marketState.lexState.lastSqrtPriceX96) = LatentMath
                .getMarketStateFromLiquidityAndDebt(
                    _edgeSqrtPriceX96_A,
                    _edgeSqrtPriceX96_B,
                    marketState.liquidity,
                    marketState.dexAmounts[0]
                );
// ...
```

- The `lastSqrtPriceX96`, `dexAmounts`, and `liquidity` are also used inside `mint`, `redeem`, and `swap` operations, causing these operations to return incorrect token amounts.
- Full-redeem underflow/DoS via protocol fee accrual, full redeem sets `amountOut = baseTokenSupply`, but `Covenant then subtracts both` amountOut `and` protocolFees`, underflowing and reverting when any fee > 0.

```
// ...
if (aTokenAmountIn == marketState.supplyAmounts[1] && zTokenAmountIn ==
marketState.supplyAmounts[0]) {
    amountOut = marketState.baseTokenSupply;
    nextSqrtPriceX96 = _targetSqrtPriceX96;
}
// ..
```

```
// ...
        // Update market state (storage)
        marketState[redeemParams.marketId].baseSupply = baseSupply - amountOut - protocolFees;
        if (protocolFees > 0) marketState[redeemParams.marketId].protocolFeeGrowth +=
protocolFees;
// ...
```

Recommendations

Decrease `baseTokenSupply` by the calculated fee before using it in subsequent operations.

Medium findings

[M-01] `SynthToken` returns incorrect name, symbol, and decimals

Severity

Impact: Medium

Likelihood: Medium

Description

SynthToken incorrectly attempts to override inherited ERC20 state variables by declaring new private variables `_name` and `_symbol`. This creates shadowing issues rather than proper overrides.

```
contract SynthToken is ERC20, ISynthToken {
    ...
    // Variables
    string private _name; // Creates NEW variable, doesn't override ERC20's _name
    string private _symbol; // Creates NEW variable, doesn't override ERC20's _symbol

    constructor(
        address covenantCore_,
        address lexCore_,
        MarketId marketId_,
        IERC20 baseAsset_,
        AssetType synthType_
    ) ERC20("", "") { // Sets ERC20's _name and _symbol to empty strings
```

It causes `name` and `symbol` functions return empty strings. These functions access ERC20's variables (set to ""), not SynthToken's new variables.

SynthToken storage layout:

- **_balances** — mapping(address => uint256), slot 0
- **_allowances** — mapping(address => mapping(address => uint256)), slot 1
- **_totalSupply** — uint256, slot 2
- **_name** — string, slot 3
- **_symbol** — string, slot 4
- **_name** — string, slot 5
- **_symbol** — string, slot 6

The contract also `decimals` returns 18 instead of the base asset's decimals because no function override exists. It can lead to pricing error when combining with oracle price calculation.

Recommendations

It's recommended to override the getter functions to return correct values:

```
function name() public view override returns (string memory) {
    return _name;
}

function symbol() public view override returns (string memory) {
    return _symbol;
}

function decimals() public view override returns (uint8) {
    return _decimals;
}
```

[M-02] Market DoS when `baseTokenSupply` is 0 while Synth token supply is non-zero

Severity

Impact: High

Likelihood: Low

Description

Inside the `_calculateMarketState` function, which is called on every operation to ensure the market state is updated, there is a `baseTokenSupply == 0` check to ensure the market never enters a bad state:

```
    function _calculateMarketState(
        MarketId marketId,
        MarketParams calldata marketParams,
        uint256 baseTokenSupply
    ) internal view returns (LexFullState memory marketState) {
        // ...

        // calculate values if market has liquidity
        if (marketState.supplyAmounts[0] > 0 || marketState.supplyAmounts[1] > 0) {
            // if aTokenSupply or zTokenSupply > 0, this is an invalid state.
            // Block the market (which in any case, has no liquidity)

>>>         if (baseTokenSupply == 0) revert LSErrors.E_LEX_InvalidMarketState();
        // ...
}
```

However, this state, where the Synth token supply is non-zero but `baseTokenSupply` is 0 can occur, especially in the early stages of the market.

PoC :

Add the following test to `LatentSwapLEX.t.sol` :

```
    function test_PoC_DoSMarket() external {
        UnderCollateralizedTestParams memory params;

        // Set initial supply
        params.initialSupply = 1e9;

        // deploy latentSwapLEX liquid
        MockLatentSwapLEX latentSwapLEX = new MockLatentSwapLEX(
            address(this),
            P_MAX,
            P_MIN,
            P_TARGET,
            P_LIM_H,
            P_LIM_L,
            10 ** 18,
            DURATION,
            100
        );

        // Initialize
        MarketId marketId = MarketId.wrap(keccak256("Random market (LatentSwap does not
verify)"));
        MarketParams memory marketParams = MarketParams({
            baseToken: _mockBaseAsset,
            quoteToken: _mockQuoteAsset,
            oracle: _mockOracle,
            lex: address(latentSwapLEX)
        });
        latentSwapLEX.initMarket(marketId, marketParams, 100, hex"");

        // Step 1: Mint initial position
        (params.initialATokenSupply, params.initialZTokenSupply, , ) = latentSwapLEX.mint(
            MintParams({
                marketId: marketId,
                marketParams: marketParams,
                baseAmountIn: params.initialSupply,
                to: address(this),
                minATokenAmountOut: 0,
                minZTokenAmountOut: 0
            }),
            address(this),
            0
        );
        console.log("initial supply A : ", params.initialATokenSupply);
        console.log("initial supply Z : ", params.initialZTokenSupply);

        uint256 timeDelta = (((params.initialSupply) * 1 / 100) + 5) * (365 days * 1e4) / (100
* params.initialSupply) + 1;

        skip(timeDelta);
        (uint256 liquidityOut, uint128 protocolFees , ) = latentSwapLEX.redeem(
            RedeemParams({
                marketId: marketId,
                marketParams: marketParams,
                aTokenAmountIn: params.initialATokenSupply - 2,
```

```
                zTokenAmountIn: params.initialZTokenSupply,
                to: address(this),
                minAmountOut: 0
            }),
            address(this),
            params.initialSupply
        );

        vm.expectRevert(abi.encodeWithSignature("E_LEX_InvalidMarketState()"));
        bool isUnderCollateralized = latentSwapLEX.isUnderCollateralized(marketId,
marketParams, params.initialSupply - liquidityOut - protocolFees);
    }
```

It can be observed that by depositing into the market and leaving a dust amount of Synth tokens, the combination of swap fees and protocol fees can result in a dust amount of Synth tokens remaining while `baseTokenSupply` becomes 0.

Recommendations

Either prevent this scenario by checking the state after each operation, or take a small amount of Synth shares from the first depositor and mint them to a dead address to avoid this scenario.

[M-03] ETWAP logic allows grief attacks on redeem flow

Severity

Impact: Medium

Likelihood: Medium

Description

The ETWAP's implementation is used to ensure that users can redeem approximately 25% every 1 hour. The `_calculateMarketState()` updates the `lastETWAPBaseSupply` logically when the base token supply drops and sets it immediately when `baseTokenSupply` is greater than or equal to `lastETWAPBaseSupply`:

```
            // If baseTokenSupply has decreased since the last update, then decrease the
ETWAPBaseSupply tracker
        if (marketState.baseTokenSupply < marketState.lexState.lastETWAPBaseSupply)
                marketState.lexState.lastETWAPBaseSupply =
                    marketState.lexState.lastETWAPBaseSupply.rayMul(updateFactor) +
                    marketState.baseTokenSupply.rayMul(WadRayMath.RAY - updateFactor); // eTWAP
update if baseSupply decreased vs last update           <<@
```

```
        // if baseTokenSupply has increased (even in same timeblock), update record
        if (marketState.baseTokenSupply >= marketState.lexState.lastETWAPBaseSupply)
            marketState.lexState.lastETWAPBaseSupply = marketState.baseTokenSupply; //
immediatedly update if baseSupply increased vs last update               <<@
```

This `lastETWAPBaseSupply` is then passed onto `_checkRedeemCap()`, which reverts if the `remainingBaseSupply` is smaller than 75% of `eTWAPBaseTokenSupply`:

```
    function _checkRedeemCap(
        uint256 marketBaseTokenSupply,
        uint256 eTWAPBaseTokenSupply,
        uint256 redeemAmount
    ) internal pure virtual {
        // OK to redeem any amount if marketBaseTokenSupply < MAX_REDEEM_NO_CAP
        // ie, for small markets there is no limit on how much can be redeemed.
        // But for bigger markets, approx. 25% can be redeemed every 1hr
        // (this is based on ETWAP_MIN_HALF_LIFE and MAX_REDEEM_FACTOR_CAP values)
        if (eTWAPBaseTokenSupply > MAX_REDEEM_NO_CAP) {
            uint256 remainingBaseSupply;
            unchecked {
                remainingBaseSupply = (marketBaseTokenSupply > redeemAmount) ?
marketBaseTokenSupply - redeemAmount : 0;
            }
            if (remainingBaseSupply < eTWAPBaseTokenSupply.percentMul(MAX_REDEEM_FACTOR_CAP))
                revert LSErrors.E_LEX_RedeemCapExceeded();
        }
    }
```

However, the logic used here can be gamed by using the following attack:

1. A legitimate user holds some synth tokens that they would like to redeem.
2. The attacker can front-run such a transaction by leveraging a flash loan, calling the `mint()` function using a large amount, and immediately redeeming them in the same transaction.
3. This attack would spike the `eTWAPBaseTokenSupply` and, when orchestrated correctly, deny legitimate users from making a redemption till the `eTWAPBaseTokenSupply` normalises back, which would take several minutes.

This can be done easily on smaller markets and grief users for a brief period of time, and, if done methodically during volatility on larger markets, it can DoS a large chunk of users from redeeming.

Recommendations

This issue arises due to the eTWAP logic considering only the base supply plunge; it is recommended to apply the same in both directions.

[M-04] Redeem cap bypassed using `swap()`

Severity

Impact: Medium

Likelihood: Medium

Description

A critical vulnerability exists where the `swap()` function can be used to bypass the `_checkRedeemCap()` safety mechanism, which is designed to limit large, rapid withdrawals of the base asset over short periods. This allows a user to obtain baseToken via `swap()` without triggering the redeem-cap checks that `redeem()` enforces.

The `redeem()` function in `LatentSwapLEX.sol` properly calls `_checkRedeemCap()` to limit the amount of base assets that can be withdrawn over a short period, protecting the market's stability.

However, the `swap()` function, when used to swap a synth token (aToken or zToken) for the `baseToken`, performs a functionally identical operation by calling `LatentMath.computeRedeem` internally. Crucially, the `swap` function's code path **does not** include a call to `_checkRedeemCap()`.

Vulnerable Code Path:

1. A user calls `Covenant.sol::swap()` with `assetIn` as `DEBT` (zToken) or `LEVERAGE` (aToken) and `assetOut` as `BASE`.
2. This calls `LatentSwapLEX.sol::swap()`.
3. Inside `swap()`, the logic enters the block at line 795: `else if ((assetOut == AssetType.BASE) && IsExactIn)`.
4. This block calculates the amount of base token to send out by calling `LatentMath.computeRedeem`, effectively performing a redemption.
5. **No call to `_checkRedeemCap()` is ever made.**

Bypassing the redeem cap itself is not necessarily equivalent to enabling a "bank run" — the redeem cap is not intended to prevent normal liquidity outflows. However, this bypass can enable complex atomic sequences that the protocol intended to be limited. In particular, sequences such as **large mint → swap (to obtain baseToken) → redeem** executed within a single block could be used to manipulate on-chain state or prices. Given known precision limitations (for example in `sqrtPriceX96`), such atomic sequences increase the risk of temporary price discrepancies, manipulation, or causing markets to hit LTV limits unexpectedly.

The provided Proof of Concept demonstrates this flaw: a large `redeem()` transaction is correctly reverted with `E_LEX_RedeemCapExceeded`, while an equivalent `swap()` transaction for an even larger amount succeeds, bypassing the protocol's redeem-cap protection in that execution path.

This bypass undermines the intended control provided by the redeem cap and increases the risk of manipulation or LTV-triggered effects under certain conditions.

Proof of Concept

Foundry PoC:

Please copy the following POC in `CovenantLiquid.t.sol`

```
import {LSErrors} from "../src/lex/latentSwap/libraries/LSErrors.sol";
import "forge-std/console.sol";

function test_POC_redeem_failed_swap_successued() external {
    Covenant covenantCore;
    address lexImplementation;
    MarketId marketId;
    uint256 baseAmountIn = 10 * 10 ** 18;

    // deploy covenant liquid
    covenantCore = new Covenant(address(this));

    // deploy lex implementation
    lexImplementation = address(
        new LatentSwapLEX(
            address(covenantCore),
            P_MAX,
            P_MIN,
            P_TARGET,
            P_LIM_H,
            P_LIM_L,
            DISCOUNT_BALANCED,
            DURATION,
            SWAP_FEE
        )
    );

    // authorize lex
    covenantCore.setEnabledLEX(lexImplementation, true);

    // authorize oracle
    covenantCore.setEnabledOracle(_mockOracle, true);

    uint8 newFee = 255; // 2.55%
    covenantCore.setDefaultFee(newFee);
    //set base token e.g;WETH price to $2k
    MockOracle(_mockOracle).setPrice(2000 * (10 ** 18));

    // init market
    MarketParams memory marketParams = MarketParams({
        baseToken: _mockBaseAsset,
        quoteToken: _mockQuoteAsset,
        oracle: _mockOracle,
        lex: lexImplementation
    });
    marketId = covenantCore.createMarket(marketParams, hex"");
    // approve transferFrom
    IERC20(_mockBaseAsset).approve(address(covenantCore), baseAmountIn);

    //user 01 mint first

    (uint256 aTokenAmount_user01, uint256 zTokenAmount_user01) = covenantCore.mint(
        MintParams({
            marketId: marketId,
            marketParams: marketParams,
```

```
            baseAmountIn: baseAmountIn,//10 WETH
            to: address(this),
            minATokenAmountOut: 0,
            minZTokenAmountOut: 0
        })
    );

    //user 02 mint after 30 s
    vm.warp(block.timestamp + 30 );

    address user_02 = address(0x123);
    IERC20(_mockBaseAsset).transfer(user_02, baseAmountIn);
    vm.startPrank(user_02);
    IERC20(_mockBaseAsset).approve(address(covenantCore), baseAmountIn);

    (uint256 aTokenAmount_user02, uint256 zTokenAmount_user02) = covenantCore.mint(
        MintParams({
            marketId: marketId,
            marketParams: marketParams,//10 WETH
            baseAmountIn: baseAmountIn / 2,
            to: address(user_02),
            minATokenAmountOut: 0,
            minZTokenAmountOut: 0
        })
    );
    vm.stopPrank();
    //Update State - after 15 days and price drop
    vm.warp(block.timestamp + 15 days );

    // change price from $2k to $1.2k
    MockOracle(_mockOracle).setPrice(1200 * (10 ** 18));
    covenantCore.updateState(marketId, marketParams);

    //redeem
    uint256 baseAmountOut = covenantCore.redeem(
        RedeemParams({
            marketId: marketId,
            marketParams: marketParams,
            aTokenAmountIn: aTokenAmount_user01/3,
            zTokenAmountIn: aTokenAmount_user02/3,
            to: address(this),
            minAmountOut: 0
        }));


    //User ask to burn `aTokenAmount_user02 / 2` of zTokens.
    //Tx will revert, because of the redeem cap exceeded in lex contract
    vm.expectRevert(LSErrors.E_LEX_RedeemCapExceeded.selector);

    //failed redeem
    baseAmountOut = covenantCore.redeem(
        RedeemParams({
            marketId: marketId,
            marketParams: marketParams,
            aTokenAmountIn: 0,
            zTokenAmountIn: aTokenAmount_user02 / 2,
            to: address(this),
            minAmountOut: 0
```

```
            }));

        //User ask to burn `aTokenAmount_user02` of zTokens.
        //swap executed successfully because there is no redeem cap in sawp function
        uint256 swapAmountOut =  covenantCore.swap(
            SwapParams({
                marketId: marketId,
                marketParams: marketParams,
                assetIn: AssetType.DEBT,
                assetOut: AssetType.BASE,
                to: address(this),
                amountSpecified: aTokenAmount_user02 , //zTokens
                amountLimit: 0,
                isExactIn: true
            })
        );
       console.log("Base Token recived by user -AFTER SWAP-:",uint256(swapAmountOut));

    }
```

Recommendations

To fix this vulnerability, the `_checkRedeemCap()` check must be integrated into the `swap()` function's execution path whenever the `assetOut` is the `baseToken`.

[M-05] Early market can be DoSed by over-minting debt

Severity

Impact: Medium

Likelihood: Medium

Description

When the market is still in its early stages with a low `baseTokenSupply`, an attacker can mint a large proportion of zTokens close to the max debt limit. This makes the market highly sensitive to price changes, once the price drops, the market becomes undercollateralized, subsequent mint operations revert, and honest users are effectively DoS'ed from adding base liquidity and bootstrapping the market.

```
    function mint(
        MintParams calldata mintParams,
        address,
        uint256 baseTokenSupply
    )
        external
        onlyCovenantCore
        returns (uint256 aTokenAmountOut, uint256 zTokenAmountOut, uint128 protocolFees,
TokenPrices memory tokenPrices)
    {
        // Calculate market state (storage read)
```

```
        LexFullState memory currentState = _calculateMarketState(
            mintParams.marketId,
            mintParams.marketParams,
            baseTokenSupply
        );

        ////////////////////////////////
        // Validate inputs

        // check if undercollateralized (reverts if so)
>>>     _checkUnderCollateralized(currentState);

        // check LTV before mint (reverts if LTV above limit.)
        _checkMaxLTV(currentState.lexState.lastSqrtPriceX96);

        ////////////////////////////////
        // Calculate mint
        (aTokenAmountOut, zTokenAmountOut) = _calcMint(currentState, mintParams.baseAmountIn);

        // ...
    }
```

PoC :

Add the following test to `LatentSwapLEX.t.sol` :

```
    function test_PoC_DoS_Mint() external {

        UnderCollateralizedTestParams memory params;

        // Set initial supply
        params.initialSupply = 1000;

        // deploy latentSwapLEX liquid
        MockLatentSwapLEX latentSwapLEX = new MockLatentSwapLEX(
            address(this),
            P_MAX,
            P_MIN,
            P_TARGET,
            P_LIM_H,
            P_LIM_L,
            10 ** 18,
            DURATION,
            0
        );

        // Initialize
        MarketId marketId = MarketId.wrap(keccak256("Random market (LatentSwap does not
verify)"));
        MarketParams memory marketParams = MarketParams({
            baseToken: _mockBaseAsset,
            quoteToken: _mockQuoteAsset,
            oracle: _mockOracle,
            lex: address(latentSwapLEX)
        });
        latentSwapLEX.initMarket(marketId, marketParams, 0, hex"");
```

```
        (params.initialATokenSupply, params.initialZTokenSupply, , ) = latentSwapLEX.mint(
            MintParams({
                marketId: marketId,
                marketParams: marketParams,
                baseAmountIn: params.initialSupply,
                to: address(this),
                minATokenAmountOut: 0,
                minZTokenAmountOut: 0
            }),
            address(this),
            0
        );

        console.log("minted A : ", params.initialATokenSupply);
        console.log("minted Z : ", params.initialZTokenSupply);

        (uint256 ZTokenOut, , ) = latentSwapLEX.swap(
            SwapParams({
                marketId: marketId,
                marketParams: marketParams,
                assetIn: AssetType.LEVERAGE,
                assetOut: AssetType.DEBT,
                to: address(this),
                amountSpecified: params.initialATokenSupply * 90 / 100,
                amountLimit: 0,
                isExactIn: true
            }),
            address(this),
            params.initialSupply
        );

        MockOracle(_mockOracle).setPrice(WadRayMath.WAD * 95 / 100);

        bool isUnderCollateralized = latentSwapLEX.isUnderCollateralized(marketId,
marketParams, params.initialSupply);
        assert(isUnderCollateralized);

        vm.expectRevert(abi.encodeWithSignature("E_LEX_ActionNotAllowedUnderCollateralized()"));
        (params.initialATokenSupply, params.initialZTokenSupply, , ) = latentSwapLEX.mint(
            MintParams({
                marketId: marketId,
                marketParams: marketParams,
                baseAmountIn: 10 ** 18,
                to: address(this),
                minATokenAmountOut: 0,
                minZTokenAmountOut: 0
            }),
            address(this),
            params.initialSupply
        );
    }
```

The PoC demonstrates a scenario where an attacker becomes the first liquidity provider in the market, then swaps a huge portion of leverage tokens for debt tokens. Once the price drops slightly, the market becomes undercollateralized, and subsequent mint operations revert.

Recommendations

Allow mint operations (or restrict them to admin) when undercollateralized, and only block z-side redemptions or swaps that would worsen the state.

Low findings

[L-01] Protocol can be DOS due to arithmetic underflow in interest accrual

In `DebtMath.calculateApproxNotionalUpdate`, when `_lnRate` is negative, the function calculates:

`WadRayMath.RAY - rate1 + rate2 - rate3`.

Where `rate1 = abs(_lnRate) * _timeDelta / _duration`.

```
function calculateApproxNotionalUpdate(
    int256 _lnRate,
    uint256 _timeDelta,
    uint256 _duration
) internal pure returns (uint256 updateMultiplier_) {
    uint256 rate1 = (uint256((_lnRate >= 0) ? _lnRate : -_lnRate) * _timeDelta) / _duration;
    uint256 rate2 = rate1.rayMul(rate1);
    unchecked {
        rate2 = rate2 / 2;
    }
    uint256 rate3 = rate2.rayMul(rate1);
    unchecked {
        rate3 = rate3 / 3;
    }

    if (_lnRate >= 0) return WadRayMath.RAY + rate1 + rate2 + rate3;
    else return WadRayMath.RAY - rate1 + rate2 - rate3;
}
```

If `_timeDelta / _duration > 1 / |_lnRate|` (`rate1` > `WadRayMath.RAY`), the subtraction `WadRayMath.RAY - rate1` will underflow and revert:

This can occur when:

- The debt price discount becomes extremely large.
- A large time delta has passed since the last update (the pool isn't currently attractive)
- The debt duration is relatively small

The protocol can become permanently unusable due to arithmetic underflow in the `calculateApproxNotionalUpdate` function, preventing any market state updates and effectively DOSing the entire system.

It's recommended to: - use `WadRayMath.RAY + rate2 - rate1 - rate3` OR - use accurate exponential calculation to completely avoid the issue.

[L-02] Inconsistent price reset behavior in full redemption scenarios

The `LatentMath.computeRedeem` function returns the current market price when all tokens are redeemed, while the higher-level `LatentSwapLEX._calcRedeem` function correctly resets to the target price in full redemption scenarios. This creates an inconsistency in price reset behavior:

```
    function computeRedeem(
        uint160 currentLiquidity,
        uint160 currentSqrtRatioX96,
        uint160 edgeSqrtRatioX96_A,
        uint160 edgeSqrtRatioX96_B,
        uint256 zTokenAmtIn,
        uint256 aTokenAmtIn
    ) internal pure returns (uint160 liquidityOut, uint160 nextSqrtRatioX96) {
        ...
        if (zTokenAmtIn >= zDexAmount && aTokenAmtIn >= aDexAmount) {
            //Full burn
            return (currentLiquidity, currentSqrtRatioX96); // @audit returns current price
        } else {
```

It's recommended to modify `LatentMath.computeRedeem` to reset to the target price in full redemption scenarios.

[L-03] Missing zero amount validation for non-base asset outputs in swaps

The `checkSwapOutputs` function in ValidationLogic only validates zero output amounts when assetOut is `AssetType.BASE`, but fails to perform the same validation when the output asset is `AssetType.LEVERAGE` or `AssetType.DEBT`. This inconsistency could allow swaps that result in zero synthetic tokens being minted, leading to poor user experience and potential loss of input tokens.

```
    function checkSwapOutputs(
        SwapParams calldata swapParams,
        uint256 baseSupply,
        uint256 amountCalculated
    ) internal pure {
        ...
        if ((swapParams.assetOut == AssetType.BASE) && (amountCalculated == 0)) revert
Errors.E_InsufficientAmount(); // @audit Only checks zero for BASE tokens
}
    }
```

It's recommended to add a zero amount check for all asset types:

```
if (amountCalculated == 0) revert Errors.E_InsufficientAmount();
```

[L-04] Missing validation for identical base and quote Tokens in market

The `checkMarketParams` function fails to validate that `baseToken` and `quoteToken` are different addresses when creating a new market. This allows the creation of markets where both tokens are identical, which would result in a dysfunctional market with meaningless price calculations and swap operations.

It's recommended to add a validation check to ensure `baseToken` and `quoteToken` are different.

[L-05] Potential division-by-zero in `_dexToSynth`

Inside `_dexToSynth`, when assetType is `AssetType.LEVERAGE`, one branch incorrectly checks `marketState.supplyAmounts[1] == 0` instead of `marketState.dexAmounts[1]`. This can cause the operation to revert if `marketState.dexAmounts[1]` is 0 due to a division-by-zero.

```
    function _dexToSynth(
        LexFullState memory marketState,
        uint256 dexAmount,
        AssetType assetType,
        Math.Rounding rounding
    ) internal pure returns (uint256 synthAmount) {
        // @dev - if asset is debt, use the debtDexRatioX96, which is the correct ratio for debt
irrespective of supply
        // for leverage token, use the actual dex + supply amounts for greater breadth of token
scales
        if (assetType == AssetType.DEBT)
            synthAmount = Math.mulDiv(dexAmount, FixedPoint96.Q96, marketState.debtDexRatioX96,
rounding);
        else if (assetType == AssetType.BASE)
            synthAmount = Math.mulDiv(dexAmount, FixedPoint96.Q96,
marketState.liquidityRatioX96, rounding);
        else if (marketState.supplyAmounts[1] == 0)
            synthAmount = dexAmount; // if no supply for leverage token, assume ratio == 1
>>>     else synthAmount = Math.mulDiv(dexAmount, marketState.supplyAmounts[1],
marketState.dexAmounts[1], rounding);
    }
```

[L-06] ETWAP check blocks the last user from immediate full withdrawal

`redeem` operation in `LatentSwapLEX` always checks the ETWAP redeem cap, even when called by the last withdrawer.

```
    function redeem(
        RedeemParams calldata redeemParams,
        address sender,
        uint256 baseTokenSupply
```

```
    ) external onlyCovenantCore returns (uint256 amountOut, uint128 protocolFees, TokenPrices
memory tokenPrices) {
        // ...

        ////////////////////////////////
        // Calculate redeem
        (amountOut, currentState.lexState.lastSqrtPriceX96) = _calcRedeem(
            currentState,
            redeemParams.aTokenAmountIn,
            redeemParams.zTokenAmountIn
        );

        ////////////////////////////////
        // Validate outputs

        _checkRedeemCap(baseTokenSupply, currentState.lexState.lastETWAPBaseSupply, amountOut);
// ...
```

This causes the last withdrawer to unnecessarily batch operations in order to perform a full withdrawal. Consider skipping the redeem cap check when the operation is a full withdrawal that empties the market.

[L-07] Preview functions fail to verify the outputs

The `previewMint()`, `previewRedeem()`, and `previewSwap()` functions enable users to preview their transaction results beforehand, allowing them to verify the outputs before submitting the actual transaction on-chain.

However, these functions fail to actually mimic the transactions as they do not verify the outputs via the 'ValidationLogic` contract:

```
        // Validate mint amounts
        ValidationLogic.checkMintOutputs(mintParams, aTokenAmountOut, zTokenAmountOut);

        // Validate redeem amounts
        ValidationLogic.checkRedeemOutputs(redeemParams, baseSupply, amountOut);

        // Validate swap amounts
        ValidationLogic.checkSwapOutputs(swapParams, baseSupply, amountCalculated);
```

Hence, a user might be tricked into believing their transaction would go through.

It is recommended to add the above validation checks to the respective preview functions.

[L-08] Incorrect concat usage corrupts token name and symbol

The SynthToken's constructor sets the token's name and symbol:

```
        // Synth tokens have the same name and symbol as underlying asset, with an additional a/
z identifier.
        _name = string.concat(_name, string((synthType_ == AssetType.DEBT) ? "z" : "a"));
        _symbol = string.concat(_symbol, string((synthType_ == AssetType.DEBT) ? "z" : "a"));
```

However, as per the docs, the `z` and `a` are supposed to be added as a prefix, whereas it's currently being added to the suffix.

It is recommended to add them to the prefix:

```
        _name = string.concat(string((synthType_ == AssetType.DEBT) ? "z" : "a"), _name);
        _symbol = string.concat(string((synthType_ == AssetType.DEBT) ? "z" : "a"), _symbol);
```

[L-09] Market state broken: `mint()` disabled but swaps creating debt allowed

A critical design flaw allows the protocol to enter a broken state where the supply of debt tokens (zTokens) is zero, but leverage tokens (aTokens) still exist. In this state, the primary liquidity provision function, `mint()`, becomes permanently disabled, while the `swap()` function can still be used to create new debt, leading to inconsistent and illogical market behavior.

This state can be reached if users, through a series of swaps (e.g., `DEBT` -> `LEVERAGE`). When `zTokenSupply` becomes zero, the market's internal price (`lastSqrtPriceX96`) is driven to its absolute minimum boundary (`_edgeSqrtPriceX96_A`).

`mint()` **Fails:** When a user subsequently attempts to call `mint()`, the `_calcMint()` function in `LatentSwapLEX` correctly calculates that zero zTokens should be minted because the price is at the boundary. This `zTokenAmountOut = 0` then causes the transaction to revert in `ValidationLogic.sol::checkMintOutputs`, which prohibits zero-amount mints. The market is now effectively "bricked" for its primary liquidity mechanism. `swap()` **Succeeds:** In this same broken state, a user can successfully call `swap()` to convert `BASE` tokens into `DEBT` (zTokens). This will make the `mint()` callable again.

This inconsistency breaks the core logic of the market. The intended mechanism for adding liquidity is blocked, forcing users into less efficient, single-sided swaps to restore the market to a functional state. This creates a confusing and fragile system that can easily become stuck in a non-functional state.

Proof of Concept Foundry PoC:

Please copy the following POC in `CovenantLiquid.t.sol`

```
    function test_POC_mint_revert_zero_ztoken() external {
     Covenant covenantCore;
     address lexImplementation;
     MarketId marketId;
     uint256 baseAmountIn = 10 * 10 ** 18;

     // deploy covenant liquid
     covenantCore = new Covenant(address(this));

     // deploy lex implementation
     lexImplementation = address(
         new LatentSwapLEX(
```

```
                address(covenantCore),
                P_MAX,
                P_MIN,
                P_TARGET,
                P_LIM_H,
                P_LIM_L,
                DISCOUNT_BALANCED,
                DURATION,
                SWAP_FEE
            )
        );

        // authorize lex
        covenantCore.setEnabledLEX(lexImplementation, true);

        // authorize oracle
        covenantCore.setEnabledOracle(_mockOracle, true);

        uint8 newFee = 255; // 2.55%
        covenantCore.setDefaultFee(newFee);
        MockOracle(_mockOracle).setPrice(2000 * (10 ** 18));
        // init market
        MarketParams memory marketParams = MarketParams({
            baseToken: _mockBaseAsset,
            quoteToken: _mockQuoteAsset,
            oracle: _mockOracle,
            lex: lexImplementation
        });

        marketId = covenantCore.createMarket(marketParams, hex"");

{
        // approve transferFrom
        IERC20(_mockBaseAsset).approve(address(covenantCore), baseAmountIn * 3);
    //console.log("======== MINT 001 ==========");

        // mint
        (uint256 aTokenAmount, uint256 zTokenAmount) = covenantCore.mint(
            MintParams({
                marketId: marketId,
                marketParams: marketParams,
                baseAmountIn: baseAmountIn,
                to: address(this),
                minATokenAmountOut: 0,
                minZTokenAmountOut: 0
            })
        );

    //console.log("======== MINT 002 - after 30 s =============");
     vm.warp(block.timestamp + 30 );

        ( uint256 aTokenAmount01,uint256 zTokenAmount01) = covenantCore.mint(
            MintParams({
                marketId: marketId,
                marketParams: marketParams,
                baseAmountIn: baseAmountIn/2,
                to: address(this),
                minATokenAmountOut: 0,
```

```
                minZTokenAmountOut: 0
            })
        );

        vm.warp(block.timestamp + 15 days );
        MockOracle(_mockOracle).setPrice(5000 * (10 ** 18));

        uint256 swapAmountOut01 =  covenantCore.swap(SwapParams({
               marketId: marketId,
               marketParams: marketParams,
               assetIn: AssetType.DEBT,
               assetOut: AssetType.LEVERAGE,
               to: address(this),
               amountSpecified: zTokenAmount01 + zTokenAmount,
               amountLimit: 0,
               isExactIn: true
     }));


    vm.warp(block.timestamp + 1 );// one second later
    covenantCore.updateState(marketId, marketParams);
 }
```

[L-10] Inconsistent LTV blocks zToken redemptions during undercollateralization

The redeem function contains inconsistent logic that prevents beneficial market improvements during undercollateralization. While the protocol correctly allows zToken-only redemptions when undercollateralized (since reducing debt improves market health), it then enforces strict LTV bounds that can block redemptions unless they're large enough to bring the market fully back within limits.

```
    function redeem(
        RedeemParams calldata redeemParams,
        address sender,
        uint256 baseTokenSupply
    ) external onlyCovenantCore returns (uint256 amountOut, uint128 protocolFees, TokenPrices
memory tokenPrices) {
        ...
        // Check whether action pushes LTV past limit
        // @dev - reverts if so.
        _checkMaxLTV(currentState.lexState.lastSqrtPriceX96); // @audit this can block
beneficial zToken redemptions that reduces the price

        ...
    }
```

When a market is undercollateralized with `currentPrice > _limHighSqrtPriceX96`: - Small zToken redemptions: Improve LTV but don't push price below `_limHighSqrtPriceX96` -> REVERT. - Large zToken redemptions: Improve LTV and push price below `_limHighSqrtPriceX96` -> SUCCESS.

This creates a problematic threshold effect where beneficial actions are blocked.

The swap function handles this correctly by only enforcing LTV checks for actions that worsen the situation.

Recommendations

Skip LTV bounds checking for zToken-only redemptions when undercollateralized.

[L-11] Inconsistent redeem cap allows excessive withdrawals

The `MAX_REDEEM_NO_CAP` constant in `LatentSwapLEX` is set to a fixed value of `10**9` base units, which creates drastically different economic thresholds for tokens with different decimal places. This allows markets to be fully drained without any rate limiting when below the threshold.

```
uint32 public constant MAX_REDEEM_NO_CAP = 10 ** 9;

function _checkRedeemCap(
    uint256 marketBaseTokenSupply,
    uint256 eTWAPBaseTokenSupply,
    uint256 redeemAmount
) internal pure virtual {
    if (eTWAPBaseTokenSupply > MAX_REDEEM_NO_CAP) {
        ...
    }
}
```

For WBTC (8 decimals), this translates to **10 BTC** (`10^9 / 10^8 = 10`), meaning any market with an ETWAP supply ≤ 10 BTC can be completely redeemed in a single transaction without triggering the redeem cap protection. For comparison: - USDC (6 decimals): 1,000 USDC threshold. - WBTC (8 decimals): 10 BTC threshold. - 18-decimal tokens: 1e-9 tokens threshold.

The redeem cap mechanism is designed to limit redemptions to 25% of the ETWAP supply per 30-minute window when active, but this protection is completely bypassed for markets below the threshold. This creates a significant vulnerability for high-value, low-decimal tokens like WBTC where 10 BTC represents substantial value that should not be withdrawable without rate limiting.

Recommendations

Implement a decimal threshold calculation to ensure consistent economic limits across all tokens.

[L-12] Incorrect 512-bit number comparison in computeLiquidity function

In the `computeLiquidity` function, the code attempts to check if `q > beta^2` where both values are 512-bit numbers represented as (`low_bits` , `high_bits`) pairs:

```
    function computeLiquidity(
        uint160 sqrtRatioX96_A,
        uint160 sqrtRatioX96_B,
        uint256 zTokenAmount,
        uint256 aTokenAmount
    ) internal pure returns (uint160) {
        ...
        if ((vars.b2X192_0 < vars.qX192_0) && (vars.b2X192_1 <= vars.qX192_1)) {
            return uint160(vars.betaX96 >> FixedPoint96.RESOLUTION);
```

The current comparison translates to: `(beta2_low < q_low) AND (beta2_high <= q_high)` However, the correct way to compare two 512-bit numbers A > B is: `(A_high > B_high) OR (A_high == B_high AND A_low > B_low)`

It misses the case: `q_high > beta2_high but q_low <= beta2_low`.

It can lead to wrong liquidation calculation in case q > beta^2.

Recommendations

It's recommended to update the comparison to cover all cases.

[L-13] Protocol fee updates cause loss of accrued fees

The `setMarketProtocolFee` function in `Covenant.sol` updates a market's protocol fee rate without first accruing the fees owed under the current rate. This causes the protocol to lose all fees that should have been collected between the last update and the fee change.

```
    function setMarketProtocolFee(MarketId marketId, uint8 newFee) external onlyOwner
noDelegateCall lock(marketId) {
        address lex = idToMarketParams[marketId].lex;
        uint8 oldFee = ILiquidExchangeModel(lex).getLexConfig(marketId).protocolFee;
        ILiquidExchangeModel(lex).setMarketProtocolFee(marketId, newFee); // @audit not accrue
fee
        emit Events.UpdateMarketProtocolFee(marketId, oldFee, newFee);
    }
```

If the fee rate is changed from 1% to 0.5% after 7 days without any intervening transactions, the protocol loses 7 days worth of fees at the 1% rate. The next update will incorrectly calculate the entire 7-day period at the new 0.5% rate instead.

Recommendations

Call `updateState()` to accrue pending fees before changing the fee rate:

```
function setMarketProtocolFee(MarketId marketId, uint8 newFee) external onlyOwner lock(marketId)
{
    ...
    // Accrue fees at current rate first
+    uint128 protocolFees = ILiquidExchangeModel(lex).updateState(
+        marketId,
+        idToMarketParams[marketId],
```

```
+         marketState[marketId].baseSupply
+     );

+     if (protocolFees > 0) {
+         marketState[marketId].protocolFeeGrowth += protocolFees;
+         marketState[marketId].baseSupply -= protocolFees;
+     }

    ILiquidExchangeModel(lex).setMarketProtocolFee(marketId, newFee);
    emit Events.UpdateMarketProtocolFee(marketId, oldFee, newFee);
}
```

[L-14] `redeem` caps inputs for calculation but burns uncapped amounts, causing reverts

In `LatentSwapLEX.redeem`, `_calcRedeem` caps `aTokenAmountIn` and `zTokenAmountIn` to available supply for the math, but the function later burns the original, uncapped `redeemParams` amounts.

```
function _calcRedeem(
        LexFullState memory marketState,
        uint256 aTokenAmountIn,
        uint256 zTokenAmountIn
    ) internal view returns (uint256 amountOut, uint160 nextSqrtPriceX96) {
        // limit amountIn to available supply
        if (aTokenAmountIn > marketState.supplyAmounts[1]) aTokenAmountIn =
marketState.supplyAmounts[1];
        if (zTokenAmountIn > marketState.supplyAmounts[0]) zTokenAmountIn =
marketState.supplyAmounts[0];
// ...
}
```

```
    function redeem(
        RedeemParams calldata redeemParams,
        address sender,
        uint256 baseTokenSupply
    ) external onlyCovenantCore returns (uint256 amountOut, uint128 protocolFees, TokenPrices
memory tokenPrices) {
  // ...
>>>     (amountOut, currentState.lexState.lastSqrtPriceX96) = _calcRedeem(
            currentState,
            redeemParams.aTokenAmountIn,
            redeemParams.zTokenAmountIn
        );

   // ...
>>>     ISynthToken(currentState.lexConfig.aToken).lexBurn(sender, redeemParams.aTokenAmountIn);
>>>     ISynthToken(currentState.lexConfig.zToken).lexBurn(sender, redeemParams.zTokenAmountIn);
// ...
}
```

This desynchronization causes redeem to revert when the caller provides amounts above the caps (e.g., race conditions), even though the contract computed a valid `amountOut` using the capped values. leads to inconsistent accounting expectations.

PoC :

Add the following test to `LatentSwapLEX.t.sol` :

```
    function test_PoC_Redeem_Revert() external {

        UnderCollateralizedTestParams memory params;

        // Set initial supply
        params.initialSupply = WadRayMath.WAD * 100; // 100 WAD

        // deploy latentSwapLEX liquid
        MockLatentSwapLEX latentSwapLEX = new MockLatentSwapLEX(
            address(this),
            P_MAX,
            P_MIN,
            P_TARGET,
            P_LIM_H,
            P_LIM_L,
            10 ** 18,
            DURATION,
            0
        );

        // Initialize
        MarketId marketId = MarketId.wrap(keccak256("Random market (LatentSwap does not
verify)"));
        MarketParams memory marketParams = MarketParams({
            baseToken: _mockBaseAsset,
            quoteToken: _mockQuoteAsset,
            oracle: _mockOracle,
            lex: address(latentSwapLEX)
        });
        latentSwapLEX.initMarket(marketId, marketParams, 0, hex"");

        (params.initialATokenSupply, params.initialZTokenSupply, , ) = latentSwapLEX.mint(
            MintParams({
                marketId: marketId,
                marketParams: marketParams,
                baseAmountIn: params.initialSupply,
                to: address(this),
                minATokenAmountOut: 0,
                minZTokenAmountOut: 0
            }),
            address(this),
            0
        );

        vm.expectRevert();
        (uint256 liquidityOut, , ) = latentSwapLEX.redeem(
            RedeemParams({
                marketId: marketId,
                marketParams: marketParams,
```

```
                aTokenAmountIn: params.initialATokenSupply + 1,
                zTokenAmountIn:  params.initialZTokenSupply + 1,
                to: address(this),
                minAmountOut: 0
            }),
            address(this),
            params.initialSupply
        );
    }
```

Recommendations

Return capped values from `_calcRedeem` and burn those amounts instead.

[L-15] `swap` operation can lead to an undercollateralized state

When `swap` operation is performed, the undercollateralization check is done before the swap is executed. However, after execution, it is possible for `dexAmounts[0]` to exceed the `maxDebtValue`, causing the market to become undercollateralized.

```
    function swap(
        SwapParams calldata swapParams,
        address sender,
        uint256 baseTokenSupply
    )
        external
        onlyCovenantCore
        returns (uint256 amountCalculated, uint128 protocolFees, TokenPrices memory tokenPrices)
    {
        // ...
        if ((swapParams.assetIn != AssetType.DEBT) || (swapParams.assetOut != AssetType.BASE))
            _checkUnderCollateralized(currentState);

        ////////////////////////////////
        // Calculate swap
        (amountCalculated, currentState.lexState.lastSqrtPriceX96) = _calcSwap(
            currentState,
            swapParams.amountSpecified,
            swapParams.assetIn,
            swapParams.assetOut,
            swapParams.isExactIn
        );

        ////////////////////////////////
        // Validate outputs

        // LTV checks
        // Do not allow aToken sales, or zToken buys if it takes market past LTV limits

        if ((swapParams.assetIn == AssetType.LEVERAGE) || (swapParams.assetOut ==
AssetType.DEBT))
            _checkMaxLTV(currentState.lexState.lastSqrtPriceX96);
// ...
}
```

PoC :

Add the following test to `LatentSwapLEX.t.sol` :

```
    function test_PoC_undercollateralizedUnchecked() external {
        UnderCollateralizedTestParams memory params;

        // Set initial supply
        params.initialSupply = WadRayMath.WAD * 100; // 100 WAD
        params.reducedBasePrice = WadRayMath.WAD * 55 / 100; // 0.55 WAD

        // deploy latentSwapLEX liquid
        MockLatentSwapLEX latentSwapLEX = new MockLatentSwapLEX(
            address(this),
            P_MAX,
            P_MIN,
            P_TARGET,
            P_LIM_H,
            P_LIM_L,
            10 ** 18,
            DURATION,
            255
        );

        // Initialize
        MarketId marketId = MarketId.wrap(keccak256("Random market (LatentSwap does not
verify)"));
        MarketParams memory marketParams = MarketParams({
            baseToken: _mockBaseAsset,
            quoteToken: _mockQuoteAsset,
            oracle: _mockOracle,
            lex: address(latentSwapLEX)
        });
        latentSwapLEX.initMarket(marketId, marketParams, 0, hex"");

        (params.initialATokenSupply, params.initialZTokenSupply, , ) = latentSwapLEX.mint(
            MintParams({
                marketId: marketId,
                marketParams: marketParams,
                baseAmountIn: params.initialSupply,
                to: address(this),
                minATokenAmountOut: 0,
                minZTokenAmountOut: 0
            }),
            address(this),
            0
        );

        // reduce price
        MockOracle(_mockOracle).setPrice(params.reducedBasePrice);


        latentSwapLEX.swap(
            SwapParams({
                marketId: marketId,
                marketParams: marketParams,
                assetIn: AssetType.LEVERAGE,
```

```
                assetOut: AssetType.DEBT,
                to: address(this),
                amountSpecified: params.initialATokenSupply * 40 / 100, // Try to swap half of
aTokens,
                amountLimit: 0,
                isExactIn: true
            }),
            address(this),
            params.initialSupply
        );

        // mocking decrease total supply by 5% fee
        bool isUnderCollateralized = latentSwapLEX.isUnderCollateralized(marketId,
marketParams, params.initialSupply - (params.initialSupply * 5 / 100));
        assertTrue(isUnderCollateralized,
"Market should be undercollateralized after base price reduction");
    }
```

From the test, incorporating swap fees and protocol fee deductions, the max LTV check after the swap is not sufficient to prevent undercollateralization.

Recommendations

Consider rechecking or moving the `isUnderCollateralized` check after the swap.

[L-16] Loss of protocol fee due to lack of decimal consideration during accrual

The protocol fee is calculated inside the `_calculateMarketState()` function whenever core protocol actions such as swap, redeem, or mint take place:

```
            if (marketState.lexConfig.protocolFee > 0) {
                uint256 fee = DebtMath.calculateLinearAccrual(
                    baseTokenSupply,
                    marketState.lexConfig.protocolFee,
                    elapsedTime
                );
                if (fee > type(uint96).max) fee = type(uint96).max;
                if (fee > (baseTokenSupply >> 3)) fee = baseTokenSupply >> 3; // @dev set max
update of 12.5% of baseTokenSupply
                unchecked {
                    marketState.accruedProtocolFee = uint96(fee);
                }
            }
```

However, upon further observation of `calculateLinearAccrual()`, it is found to have not considered fee accrual for tokens with lower decimals:

```
    function calculateLinearAccrual(
        uint256 _value,
        uint256 _rate,
        uint256 _timeDelta
```

```
    ) internal pure returns (uint256 accrualValue_) {
        return Math.mulDiv(_value, _rate * _timeDelta, SECONDS_PER_YEAR * 1e4);
    }
```

A decent impact can be observed for tokens such as WBTC (8 decimal). A hypothetical scenario would be:

1. Total base token supply reaches 1 WBTC, which can be considered a decent market size (approx. $110k at current market rate).
2. Protocol averages a transaction every 60 seconds, and the rate is set to be 100 (1%):

```
(1e8 * (1e2 * 60)) / (31536000 * 1e4) = 1.90258751903 ~ 1.   (rounding down in solidity)
```

1. For a smaller base Token supply, such as 0.5 WBTC, it would round down to zero:

```
(5e7 * (1e2 * 60)) / (31536000 * 1e4) = 0.95129375951 ~ 0    (rounding down in solidity)
```

This example considers the fee to be set as 1%, however, it can be set to 10 (0.1%) as well, which would result in an even higher protocol fee loss.

Also, an attacker can grief the protocol by updating the market state by swapping dust amounts just before the fee accrual could be significant.

Recommendations

It is recommended to use a global high-precision index in RAY, which would grow linearly as we advance it periodically upon state update. This can be used for transferring fee shares proportional to the index delta.

[L-17] Market pause does not stop interest and fee accrual

The market pause functionality, controlled by Covenant.sol::setMarketPause(), is intended to freeze a market during emergencies. However, the current implementation only blocks new transactions (`mint()` , `redeem()` , `swap()`) but fails to stop the time-based accrual of interest and protocol fees.

Interest and fee calculations in `LatentSwapLEX.sol::_calculateMarketState()` are based on the time elapsed since `lastUpdateTimestamp` . When a market is paused for an extended period and then un-paused, the very first transaction will calculate and apply interest and fees for the entire duration of the pause.

This has two severe consequences:

- Unfair Penalization: It is fundamentally unfair to charge a/zToken holders interest for a period when the market was non-functional and they were unable to manage or exit their positions.
- Market Shock: The sudden application of a large, accumulated interest payment upon un-pausing can drastically alter the market's internal pricing and LTV. This could immediately push an otherwise stable market into a high-risk or undercollateralized state, causing unexpected liquidations or preventing users from interacting as intended.

Recommendations

A "pause" should imply a complete freeze of the market's financial state, not merely a suspension of the user interactions (the same thing should be done when the market gets empty).

[L-18] The last withdrawer can be sandwiched to extract fees from the `redeem` operation

When a `redeem` is requested and it is not by the last withdrawer, the fee is taken from `amountOut` and distributed to the remaining liquidity providers.

```
    function _calcRedeem(
        LexFullState memory marketState,
        uint256 aTokenAmountIn,
        uint256 zTokenAmountIn
    ) internal view returns (uint256 amountOut, uint160 nextSqrtPriceX96) {
        // limit amountIn to available supply
        if (aTokenAmountIn > marketState.supplyAmounts[1]) aTokenAmountIn =
marketState.supplyAmounts[1];
        if (zTokenAmountIn > marketState.supplyAmounts[0]) zTokenAmountIn =
marketState.supplyAmounts[0];

        // Check for a full redeem
        if (aTokenAmountIn == marketState.supplyAmounts[1] && zTokenAmountIn ==
marketState.supplyAmounts[0]) {
            amountOut = marketState.baseTokenSupply;
            nextSqrtPriceX96 = _targetSqrtPriceX96; // @dev - when full redeem, set to target
price
        } else {
            uint256 zTokenDexIn = _synthToDex(marketState, zTokenAmountIn, AssetType.DEBT,
Math.Rounding.Floor);
            uint256 aTokenDexIn = _synthToDex(marketState, aTokenAmountIn, AssetType.LEVERAGE,
Math.Rounding.Floor);

            // Calculate liquidity out given tokens in
            uint160 liquidityOut;
            (liquidityOut, nextSqrtPriceX96) = LatentMath.computeRedeem(
                marketState.liquidity,
                marketState.lexState.lastSqrtPriceX96,
                _edgeSqrtPriceX96_A,
                _edgeSqrtPriceX96_B,
```

```
                zTokenDexIn,
                aTokenDexIn
            );

            // Calculate baseToken amount out given liquidity out
            // Round down amount out
            amountOut = _dexToSynth(marketState, liquidityOut, AssetType.BASE,
Math.Rounding.Floor);
            if (amountOut > marketState.baseTokenSupply) amountOut =
marketState.baseTokenSupply;

            // Charge fee by reducing out amount
>>>         if (_swapFee > 0) amountOut = amountOut.percentMul(PercentageMath.PERCENTAGE_FACTOR
- _swapFee);
        }
    }
```

This makes the last withdrawer vulnerable to a sandwich attack. When the last withdrawer performs a full `redeem`, an attacker can front-run the operation by depositing a small amount of liquidity. Then, when the `redeem` is executed, the fee is extracted, and the attacker can `redeem` to collect it.

PoC :

Add the following test to `LatentSwapLEX.t.sol` :

```
    function test_PoC_Swap_Sandwich() external {

        UnderCollateralizedTestParams memory params;

        // Set initial supply
        params.initialSupply = WadRayMath.WAD * 100; // 100 WAD

        // deploy latentSwapLEX liquid
        MockLatentSwapLEX latentSwapLEX = new MockLatentSwapLEX(
            address(this),
            P_MAX,
            P_MIN,
            P_TARGET,
            P_LIM_H,
            P_LIM_L,
            10 ** 18,
            DURATION,
            255
        );

        // Initialize
        MarketId marketId = MarketId.wrap(keccak256("Random market (LatentSwap does not
verify)"));
        MarketParams memory marketParams = MarketParams({
            baseToken: _mockBaseAsset,
            quoteToken: _mockQuoteAsset,
            oracle: _mockOracle,
            lex: address(latentSwapLEX)
        });
        latentSwapLEX.initMarket(marketId, marketParams, 0, hex"");
```

```
uint256 userABaseAmount = 1e9;

(params.initialATokenSupply, params.initialZTokenSupply, , ) = latentSwapLEX.mint(
    MintParams({
        marketId: marketId,
        marketParams: marketParams,
        baseAmountIn: params.initialSupply,
        to: address(this),
        minATokenAmountOut: 0,
        minZTokenAmountOut: 0
    }),
    address(this),
    0
);
// user A front-ran
(uint256 userAAToken, uint256 userAZToken, , ) = latentSwapLEX.mint(
    MintParams({
        marketId: marketId,
        marketParams: marketParams,
        baseAmountIn: userABaseAmount,
        to: address(this),
        minATokenAmountOut: 0,
        minZTokenAmountOut: 0
    }),
    address(this),
    params.initialSupply
);

uint256 lastLiq = params.initialSupply + userABaseAmount;

// "last" withdrawer full withdraw full withdraw
(uint256 liquidityOut, , ) = latentSwapLEX.redeem(
    RedeemParams({
        marketId: marketId,
        marketParams: marketParams,
        aTokenAmountIn: params.initialATokenSupply,
        zTokenAmountIn:  params.initialZTokenSupply,
        to: address(this),
        minAmountOut: 0
    }),
    address(this),
    lastLiq
);


// user A full withdraw
(uint256 liquidityAOut, , ) = latentSwapLEX.redeem(
    RedeemParams({
        marketId: marketId,
        marketParams: marketParams,
        aTokenAmountIn: userAAToken,
        zTokenAmountIn:  userAZToken,
        to: address(this),
        minAmountOut: 0
    }),
    address(this),
    lastLiq - liquidityOut
```

```
        );

        assertTrue(liquidityAOut > userABaseAmount, "User A should have profit ");
    }
```

Recommendations

Redesign the `redeem` fee to avoid this scenario. Instead of distributing it to the remaining liquidity providers, consider adding it as a protocol fee.

[L-19] Approximate swap fee used for exact_out instead of exact calculation

The `LatentSwapLex::swap()` is used for facilitating swaps, with the ability to toggle `IsExactIn` for determining if the kind of swap is exact-in or exact-out. This function at the very end charges swap fees:

```
        // Charge fee by reducing out amount (or increasing in amount)
        if (_swapFee > 0)
            calcAmount = calcAmount.percentMul(
                IsExactIn ? PercentageMath.PERCENTAGE_FACTOR - _swapFee :
PercentageMath.PERCENTAGE_FACTOR + _swapFee
            );
```

For exact-in, we are charging the fees correctly; however, for exact-out kind of swaps, using `PercentageMath.PERCENTAGE_FACTOR + _swapFee` undercharges. When we specify a desired output amount, we need to work backwards to determine the required input. Hence, `calcAmount` should be multiplied by `PercentageMath.PERCENTAGE_FACTOR / (PercentageMath.PERCENTAGE_FACTOR - _swapFee)` instead when `isExactIn` is set to false.

Also, the current logic fails to round up in case of exact-out, as users must pay at least the true required amount.

Recommendations

It is recommended to use the industry standard fee deduction method for consistent fee charging:

```
uint256 F = PercentageMath.PERCENTAGE_FACTOR; // 10_000

if (_swapFee > 0) {
    if (IsExactIn) {
        // output side should be rounded DOWN (maker-favoring)
        calcAmount = Math.mulDiv(calcAmount, F - _swapFee, F, Math.Rounding.Floor);
    } else {
        // input side must be rounded UP (payer-favoring)
        calcAmount = Math.mulDiv(calcAmount, F, F - _swapFee, Math.Rounding.Ceil);
    }
}
```