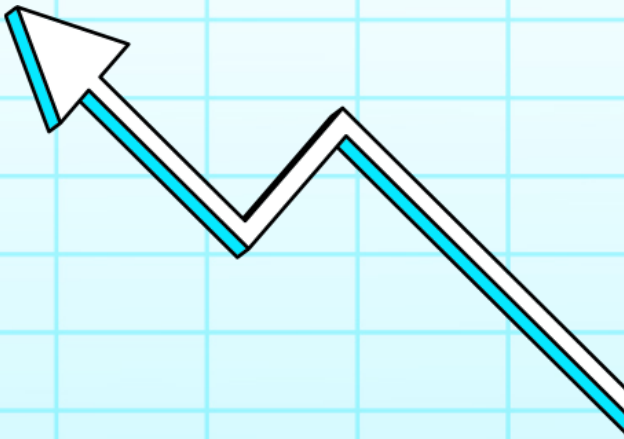
shieldify



Stacking Salmon

SECURITY REVIEW

Date: 21 March 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Stacking Salmon

Stacking Salmon is a supercharged money market where borrowers can manage Kodiak V3 positions on margin, giving market makers superior access to capital and generating higher yields for lenders. The easy-to-use lending side (Earn) is targeted at passive investors, while the borrow side (Stack) is built for sophisticated actors and offers up to 20x leverage. This system allows market-makers to run more capital-efficient strategies.

Learn more Docs

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors

- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review was conducted over the span of 25 days by the Shieldify team. The code is written professionally, and all best practices are considered. However, the security review uncovered several High- and Medium-severity issues, including incorrect chain setting assumptions, miscalculated fees, multiple front-running and DoS risks, unfair liquidations, and a few others, along with 10 Low-severity findings.

Shieldify extends its gratitude to Stacking Salmon for cooperating with all the questions our team had and strictly following the recommendations provided.

5.1 Protocol Summary

Project Name	**Stacking Salmon**
Repository	StackingSalmon
Type of Project	DeFi, Lending Protocol
Audit Timeline	25 days
Review Commit Hash	c49807ae2965cf6d121a10507a43de1d64ba1e70
Fixes Review Commit Hash	7360ac5340e328736944902f599d78bd2b35b364

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
core/src/Lender.sol	271
core/src/RateModel.sol	32
core/src/Ledger.sol	213
core/src/Borrower.sol	241
core/src/Factory.sol	190
core/src/VolatilityOracle.sol	145
helpers/src/BadDebtProcessor.sol	92
helpers/src/UniswapPositionAppraiser.sol	73
helpers/src/Liquidator.sol	61
helpers/src/OracleUpdateHelper.sol	20

periphery/src/borrower-nft/BytesLib.sol	169
periphery/src/borrower-nft/BorrowerNFT.sol	79
periphery/src/borrower-nft/ERC721Z.sol	140
periphery/src/managers/BorrowerNFTWithdrawManager.sol	14
periphery/src/managers/SimpleManager.sol	9
periphery/src/managers/UniswapNFTManager.sol	114
periphery/src/managers/FrontendManager.sol	90
periphery/src/managers/BoostManager.sol	182
periphery/src/managers/BorrowerNFTMultiManager.sol	15
periphery/src/managers/Permit2Manager.sol	32
periphery/src/managers/BorrowerNFTSimpleManager.sol	9
periphery/src/Router.sol	74
periphery/src/LenderLens.sol	16
periphery/src/BorrowerLens.sol	96
Total	**2377**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **High** issues: **2**
- **Medium** issues: **8**
- **Low** issues: **10**

ID	Title	Severity	Status
[H-01]	Incorrect Assumptions About Chain Native Token Will Heavily Skew Protocol Pricing Mechanism	High	Fixed
[H-02]	The `feeProtocol` Calculation in the `VolatilityOracle::_getPoolMetadata()` Is Flawed	High	Fixed
[M-01]	Front-Running Vulnerability in `enrollCourier()` Function	Medium	Fixed
[M-02]	Front-Running Vulnerability in `Lender::redeem()` Function Leading to Denial of Redemption	Medium	Fixed
[M-03]	Swap Callback Rejects Current Time as `deadline`	Medium	Fixed
[M-04]	Uniswap Position Fees Are Not Included in Asset Calculation For Warnings and Liquidation	Medium	Fixed
[M-05]	Incorrectly Hardcoded `_INIT_CODE_HASH` Parameter Will Lead to Wrong/Inability to Perform Appraisals	Medium	Fixed
[M-06]	Denial of Service Attacks on Router Permit Functions	Medium	Fixed
[M-07]	Lack of `deadline` Check in the `BadDebtProcessor::uniswapV3FlashCallback()` Could Lead To Loss of Funds to the User	Medium	Fixed
[M-08]	Unfair Liquidation of Healthy Accounts	Medium	Fixed
[L-01]	Signatures Have No Deadline and Cannot Be Cancelled	Low	Fixed
[L-02]	Potential Owner/Operator Backdoor in ERC721z Tokens	Low	Fixed
[L-03]	Lender Share Should Not Query Allowance from Owners	Low	Fixed
[L-04]	Missing Check for `tokenId` Existence Before Returning the `tokenURI`	Low	Fixed
[L-05]	Usage of Transfer Instead of Call For Native Funds	Low	Fixed
[L-06]	Unnecessary Allowance Buildup in the `UniswapNFTManager`	Low	Fixed
[L-07]	The Permit Functionality in `Lender` Could Not Operate with Smart Contract Multisig Wallets	Low	Fixed
[L-08]	Usage of `sqrtPriceX96` Spot Price Is Susceptible to Price Manipulation	Low	Acknowledged
[L-09]	Hardcoded and Potentially Vulnerable Slippage and Deadline Parameters	Low	Fixed
[L-10]	The `isInUse()` Function Can Provide Wrong Information	Low	Acknowledged

7. Findings

[H-01] Incorrect Assumptions About Chain Native Token Will Heavily Skew Protocol Pricing Mechanism

Severity

High Risk

Description

In various parts of the contract, the native token is meant to be used. Contracts have payable receive functions, etc. The ante constant is set to an amount that otherwise would be reasonable, were the contracts to be deployed on a chain with an expensive native token like ETH. But the contracts are to be deployed on Berachain. Its native token is BERA, which at the time of writing, costs about $6,29 per ether. Compared to ETH which costs $2371 per ether, there is a very large price discrepancy which skews protocol pricing mechanisms, disincentivizes warnings, and liquidations, encourages bad debt, etc.

The default ante is 0.01 ether, which, based on our above pricing assumptions, is about $0,06 or 6 cents. Our max ante therefore will be about $3,14.

Location of Affected Code

File: src/libraries/constants/Constants.sol

```
/// @dev The default amount of Ether required to take on debt in a `
   Borrower`. The `Factory` can override this value
/// on a per-market basis. Incentivizes calls to `Borrower.warn`.
uint208 constant DEFAULT_ANTE = 0.01 ether; //@audit Bera is much cheaper
    than eth
```

```
/// @dev The maximum amount of Ether that `Borrower`s can be required to
   post before taking on debt
uint216 constant CONSTRAINT_ANTE_MAX = 0.5 ether;
```

Impact

The main impact here revolves around protocol pricing. Since the ante determines the minimum amount required to take on debt, having it as low as $0.06 strongly incentivizes borrowing, which can quickly lead to significant bad debt. Even if governance adjusts the ante upwards, `governMarketConfig()` enforces a cap `CONSTRAINT_ANTE_MAX` of just $3—still relatively low.

On top of that, the warning mechanism isn't compelling enough; with a minimum payout of either ante/4 or the contract balance, warners won't be properly incentivized. As a result, liquidations won't happen as frequently as needed, since warnings must occur before liquidation can take place.

Recommendation

Create a governance function that can be used to update the ante and max ante values. This is a better option than hardcoding new ante values as token prices can change significantly over time.

Team Response

Fixed.

[H-02] The `feeProtocol` Calculation in the `VolatilityOracle::_getPoolMetadata()` Is Flawed

Severity

High Risk

Description

The `feeProtocol` calculation in the `VolatilityOracle::_getPoolMetadata()` is erroneous:

```
unchecked {
  if (feeProtocol % 16 != 0) metadata.gamma0 -= uint24(fee / (feeProtocol
      % 16));
  if (feeProtocol >> 4 != 0) metadata.gamma1 -= uint24(fee / (feeProtocol
      >> 4));
}
```

The `feeProtocol` is given in `uint32` and the first 16 bits are for `token1` and the lower 16 bits are for `token0`.

But the above calculation uses only 8 bits for the `protocolFee` calculation, similar to UniswapV3 but that is wrong with regard to Kodiak.

Furthermore, the `KodiakV3Factory` contract is deployed at `0xD84CBf0B02636E7f53dB9E5e45A616E05d710990` in Berachain.

In the constructor, the `defaultFeeProtocol` is given as:

```
defaultFeeProtocol = 229379500; //35%, 35%
```

Hence, this is equal to 3500, 3500 after converting to hexadecimal and then again to decimal for two tokens (token1 and token0).

Hence the `feeProtocol` is given in `basis points` and not as a `denominator` as with `UniswapV3 Pools`.

Location of Affected Code

File: src/VolatilityOracle.sol

```
unchecked {
    if (feeProtocol % 16 != 0) metadata.gamma0 -= uint24(fee / (
       feeProtocol % 16));
    if (feeProtocol >> 4 != 0) metadata.gamma1 -= uint24(fee / (
       feeProtocol >> 4));
}
```

Impact

This will lead to wrong protocol fee calculation during the new market creation and these erroneous values will result in providing wrong Implied Volatility values in the `VolatilityOracle`. This will result in erroneous price values being given as output of the `Borrower::getPrices()` function. Hence the critical function execution in the `Borrower` contract, which uses the `getPrices()` such as the `modify()`, `liquidate()`, `warn()` and `clear()`, will be broken.

Recommendation

Hence it is recommended to change the `feeProtocol` calculation logic in the `VolatilityOracle::_getPoolMetadata()` to use the entire `uint32` value. And consider the decoded fee value as a basis point and not as the denominator of the total fee.

Team Response

Fixed.

[M-01] Front-Running Vulnerability in `enrollCourier()` Function

Severity

Medium Risk

Description

The `enrollCourier()` function allows users to enroll in a referral program by associating their address with a unique `courierId` and a `cut` value, which represents the portion of the interest they will receive, from the interest earned by the referrer. The function includes a requirement that the `cut` for a given `courierId` must be zero before enrollment as shown below:

```
require(couriers[id].cut == 0);
```

The above check is in place to ensure that once a particular `id` has been enrolled its information can not be changed. This requirement further ensures that each `courierId` can only be used once. However, it also opens up the possibility for an attacker to front-run a `enrollCourier()` transaction. By monitoring the network for transactions attempting to enroll a specific `courierId`, an attacker can quickly submit their own transaction with the same id and a `cut > 0` value, causing the original transaction to revert since the `courierId` is already taken.

Since the protocol is implemented on the Berachain this vulnerability is easy to exploit due to low gas fees.

Location of Affected Code

File: Factory.sol

```
function enrollCourier(uint32 id, uint16 cut) external {
// Requirements:
// - `id != 0` because 0 is reserved as the no-courier case
// - `cut != 0 && cut < 10_000` just means between 0 and 100%
    require(id != 0 && cut != 0 && cut < 10_000);
// Once an `id` has been enrolled, its info can't be changed
    require(couriers[id].cut == 0);

    couriers[id] = Courier(msg.sender, cut);
    emit EnrollCourier(id, msg.sender, cut);
}
```

Impact

Denial of Service: Genuine users can be prevented from enrolling with their desired `ourierId`, as the attacker can claim the id first. Loss of Incentives: Users may lose the opportunity to earn incentives through the referral program if they cannot register.

Recommendation

Hence it is recommended to maintain the `courierId` in the state of the `Factory` contract and increment it for each new `enrollCourier()` request. If the `courierId` needs to be provided by the `msg.sender` himself, then it is recommended to limit the number of `enrollCourier` requests a particular address can make by either imposing a limit per user or else by charging a small fee.

Team Response

Fixed.

[M-02] Front-Running Vulnerability in `Lender::redeem()` Function Leading to Denial of Redemption

Severity

Medium Risk

Description

The `Lender::redeem()` function is used to burn `shares` from the `owner` and send the `amount` of underlying tokens to the `receiver`. In the `redeem` function, after the respective `shares` are burnt, the following check is performed on the `cache.totalSupply` value.

```
cache.totalSupply = _burn(owner, shares, inventory, cache.totalSupply);
require(cache.totalSupply == 0 || cache.totalSupply > 1e5);
```

The transaction will only proceed if the `cache.totalSupply` is either `0` or `>1e5`. Now let's consider the following scenario:

1. User A has `amount1` shares and the `totalSupply is also equal to amount1`.
2. User A decides to `redeem` his `amount1` shares and calls the `Lender::redeem()` function.
3. The transaction should proceed since now the `cache.totalSupply == 0`.
4. But an attacker `front-runs` the `redeem Tx` and deposits an amount equaling `1 wei` of shares into the `Lender` contract.
5. Now `User A` Tx executes and after the `_burn` execution the `cache.totalSupply == 1` and it is `!= 0 and < 1e5`. Hence the `UserA` redeems Tx reverts.

Location of Affected Code

File: Lender.sol

```
cache.totalSupply = _burn(owner, shares, inventory, cache.totalSupply);
require(cache.totalSupply == 0 || cache.totalSupply > 1e5);
```

Impact

A genuine user is unable to redeem his total amount of shares when he is the final `redeemer` of the `Lender` contract, even though the protocol logically allows the redemption. This will degrade the user experience and will waste gas due to revert if the user calls the `redeem` multiple times. The user will either have to redeem fewer shares than his total eligible share amount or wait for subsequent deposits to increase the `totalSupply > 1e5` (after the user redemption is accounted for). This is a loss of opportunity for the user.

Recommendation

The following recommendations can be given to resolve this issue:

Implement a Minimum Share Requirement: Instead of checking the total supply, enforce a minimum share requirement for deposits. This would prevent dust amounts from being deposited in the first place, thus mitigating the issue without affecting redemptions.

Team Response

Fixed.

[M-03] Swap Callback Rejects Current Time as `deadline`

Severity

Medium Risk

Description

In `FrontendManager.sol`'s `callback()` function, if the action is 6, a swap action is supposed to take place. The function, however, rejects the `deadline` check if the current block time is less than the `deadline`.

Location of Affected Code

File: FrontendManager.sol

```
if (action == 6) {
    (int256 amount0, int256 amount1, uint32 deadline) = abi.decode(args[i
       ], (int256, int256, uint32));
    require(block.timestamp < deadline, "deadline");
// code
}
```

Impact

This violates the common DeFi practice of including the current time with the `deadline`. Also, since the function is used in `Borrower.sol`'s `liquidate` function, the caller may be a bot, keeper or a contract programmed to scan for borrowers to liquidate. Many of these external contracts are designed to use the current `block.timestamp` as `deadline`, therefore, this check will break integration.

Recommendation

Consider using the `<=` operator instead.

Team Response

Fixed.

[M-04] Uniswap Position Fees Are Not Included in Asset Calculation For Warnings and Liquidation

Severity

Medium Risk

Description

Uniswap LPs earn fees for providing liquidity to Uniswap pools. These fees are, however, not taken as a part of the position's assets when the assets are being tallied for warning via the `_getAssets()` function. The function only returns the borrower's balance and pool liquidity at price points a and b as seen in the `Assets` struct.

Location of Affected Code

File: Borrower.sol

```
function warn(uint40 oracleSeed) external {
    uint256 slot0_ = slot0;
// Essentially `slot0.state == State.Ready && slot0.warnTime == 0`
    require(slot0_ & (SLOT0_MASK_STATE | SLOT0_MASK_AUCTION) == 0);

// Fetch prices from oracle
    (Prices memory prices, , , uint208 ante) = getPrices(oracleSeed);
// Tally assets
@>    Assets memory assets = _getAssets(slot0_, prices);
// Fetch liabilities from lenders
    (uint256 liabilities0, uint256 liabilities1) = getLiabilities();
// Ensure only unhealthy accounts get warned and liquidated
    require(!BalanceSheet.isHealthy(prices, assets, liabilities0,
       liabilities1), "Stacking Salmon: healthy");

// Start auction
    slot0 = slot0_ | (block.timestamp << 208);
    emit Warn();

    SafeTransferLib.safeTransferETH(msg.sender, address(this).balance.min
       (ante / 4));
}

function _getAssets(uint256 slot0_, Prices memory prices) private view
   returns (Assets memory assets) {
    assets.amount0AtA = assets.amount0AtB = TOKEN0.balanceOf(address(this
       ));
    assets.amount1AtA = assets.amount1AtB = TOKEN1.balanceOf(address(this
       ));

    int24[] memory positions = extract(slot0_);
    uint256 count = positions.length;
    unchecked {
        for (uint256 i; i < count; i += 2) {
// Load lower and upper ticks from the `positions` array
            int24 l = positions[i];
            int24 u = positions[i + 1];
// Fetch amount of `liquidity` in the position
@>              (uint128 liquidity, , , , ) = UNISWAP_POOL.positions(
   keccak256(abi.encodePacked(address(this), l, u)));
```

```
            if (liquidity == 0) continue;

// Compute lower and upper sqrt ratios
            uint160 L = TickMath.getSqrtRatioAtTick(l);
            uint160 U = TickMath.getSqrtRatioAtTick(u);

            uint256 amount0;
            uint256 amount1;
// Compute what amounts underlie `liquidity` at both probe prices
            (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
               prices.a, L, U, liquidity);
            assets.amount0AtA += amount0;
            assets.amount1AtA += amount1;
            (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
               prices.b, L, U, liquidity);
            assets.amount0AtB += amount0;
            assets.amount1AtB += amount1;
        }
    }
}
```

File: Borrower.sol

```
struct Assets {
// `TOKEN0.balanceOf(borrower)`, plus the amount of `TOKEN0` underlying
   its Uniswap liquidity at `Prices.a`
    uint256 amount0AtA;
// `TOKEN1.balanceOf(borrower)`, plus the amount of `TOKEN1` underlying
   its Uniswap liquidity at `Prices.a`
    uint256 amount1AtA;
// `TOKEN0.balanceOf(borrower)`, plus the amount of `TOKEN0` underlying
   its Uniswap liquidity at `Prices.b`
    uint256 amount0AtB;
// `TOKEN1.balanceOf(borrower)`, plus the amount of `TOKEN1` underlying
   its Uniswap liquidity at `Prices.b`
    uint256 amount1AtB;
}
```

File: Borrower.sol

```
function liquidate(ILiquidator callee, bytes calldata data, uint256
   closeFactor, uint40 oracleSeed) external {
        require(0 < closeFactor && closeFactor <= 10000, "Stacking Salmon
           : close");

        uint256 slot0_ = slot0;
// Essentially `slot0.state == State.Ready && slot0.warnTime > 0`
        require(slot0_ & SLOT0_MASK_STATE == 0 && slot0_ &
           SLOT0_MASK_AUCTION > 0);
        slot0 = slot0_ | (uint256(State.Locked) << 248);

// Withdraw all Uniswap positions
@>      _uniswapWithdraw(slot0_);

// Fetch prices from oracle
        (Prices memory prices, , , ) = getPrices(oracleSeed);
// Tally assets
@>      (uint256 assets0, uint256 assets1) = (TOKEN0.balanceOf(address(
   this)), TOKEN1.balanceOf(address(this)));
// Fetch liabilities from lenders
        (uint256 liabilities0, uint256 liabilities1) = getLiabilities();
// Sanity check
        {
            (uint160 sqrtPriceX96, , , , , , ) = UNISWAP_POOL.slot0();
            require(prices.a < sqrtPriceX96 && sqrtPriceX96 < prices.b);
        }

        AuctionAmounts memory amounts = BalanceSheet.
           computeAuctionAmounts(
            prices.c,
@>          assets0,
@>          assets1,
            liabilities0,
            liabilities1,
            BalanceSheet.auctionTime((slot0_ & SLOT0_MASK_AUCTION) >>
               208),
            closeFactor
        );
```

Impact

Since the user's fees are not considered, a borrower could technically have enough assets, both in liquidity and fees, but will still be deemed unhealthy and prime for liquidation.

Worse, when a user is liquidated, their Uniswap positions are burned, and their fees are collected to Borrower.sol where it is eventually considered as part of the assets to put up for auction, in a way, sending the fees to the liquidators instead.

Recommendation

It is recommended that users claim and account for their fees before evaluating their liquidation eligibility.

Team Response

Fixed.

[M-05] Incorrectly Hardcoded `_INIT_CODE_HASH` Parameter Will Lead to Wrong/Inability to Perform Appraisals

Severity

Medium Risk

Description

The `Uniswap.sol` incorrectly stores the `_INIT_CODE_HASH` as `0xe34f199b19b2b4f47f68442619d555527d244f78a3297ea89325f843f87b8b54`. This is incorrect as the protocol integrates with KodiakV3 which has a different init code hash.

From the docs, the init code hash for v3 is `0xd8e2091bc519b509176fc39aeb148cc8444418d3ce260820edc44e806c2c2339`.

Location of Affected Code

File: src/libraries/Uniswap.sol

```
bytes32 private constant _INIT_CODE_HASH = 0
    xe34f199b19b2b4f47f68442619d555527d244f78a3297ea89325f843f87b8b54; //
    @audit incorrect for kodiak
```

Impact

Pool address computation will always be incorrect - see the `computePoolAddress()` function in `Uniswap.sol`. As a result, `getAppraisal()` will either return appraisal for an incorrect pool address or revert if the returned address doesn't correspond to a pool. This will also do the `getFormattedAppraisal()` function which depends on the `getAppraisal()` function.

Recommendation

Update the `_INIT_CODE_HASH` parameter in `Uniswap.sol` to `0xd8e2091bc519b509176fc39aeb148cc8444418d3ce260820edc44e806c2c2339`.

Team Response

Fixed.

[M-06] Denial of Service Attacks on Router Permit Functions

Severity

Medium Risk

Description

The functions in `Router.sol` are the two `depositWithPermit2()` functions and the `repayWithPermit2()` function. These functions with the exception of one `depositWithPermit2()` proceeds in 2 steps:

1. Transferring the user-specified amount to Lender.sol via the `permitTransferFrom()` method.
2. Depositing/repaying the amount for the user.

The first `depositWithPermit2()` function adds an extra step which is:

1. Granting max approval to `Router.sol` using `permit` in `Lender.sol`.

Since the `permit()` function in `Lender.sol` and `permitTransferFrom` in Permit2.sol are both external,

An attacker can effectively halt these functions by monitoring the calls to obtain the various owner, spender, value, deadline, signature parameters and front-run the call, directly calling the `permit()` / `permitTransferFrom` functions in their respective contracts.

The same can be observed in the `processWithPermit()` function too, which calls `permit` in Lender.sol before calling `process`

Location of Affected Code

File: src/Router.sol

```
function depositWithPermit2(
    Lender lender,
    uint256 amount,
    uint16 transmittance,
    uint256 nonce,
    uint256 deadline,
    bytes calldata signature,
    uint32 courierId,
    uint8 v,
    bytes32 r,
    bytes32 s
) external payable returns (uint256 shares) {
    lender.permit(msg.sender, address(this), type(uint256).max, deadline,
        v, r, s);

// code
}
```

File: src/Lender.sol

```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
// code
}
```

File: src/BadDebtProcessor.sol

```
function processWithPermit(//processes bad debt
    Lender lender,
    Borrower borrower,
    IUniswapV3Pool flashPool,
    uint256 slippage,
    uint256 allowance,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    lender.permit(msg.sender, address(this), allowance, deadline, v, r, s
        );
    process(lender, borrower, flashPool, slippage);
}
```

File: Permit2.sol

```
function permitTransferFrom(
    PermitTransferFrom memory permit,
    SignatureTransferDetails calldata transferDetails,
    address owner,
    bytes calldata signature
) external {
    _permitTransferFrom(permit, transferDetails, owner, permit.hash(),
        signature);
}
```

Impact

An attacker running `permit()` with the same parameters as a legitimate call will cause approval to be given to `Router.sol` / `BadDebtProcessor.sol` while advancing the nonce.

By the time the user's legitimate call is processed, the nonce will be different and as such the calls will fail. Therefore, the function will fail and other steps of the function will be non-executable effectively dossing the process.

This can be arguably more impactful in `repayWithPermit2` because Lender.sol::repay can be time

sensitive in case a user has been warned and it requires that tokens be transferred to the contract before repayment execution.

```
  _save(cache, /* didChangeBorrowBase: */ true);

// Ensure tokens are transferred
>>  require(cache.lastBalance <= asset().balanceOf(address(this)), "
   Stacking Salmon: insufficient pre-pay");

  emit Repay(msg.sender, beneficiary, amount, units);
}
```

Proof of Concept

See here for a deeper dive.

Recommendation

Wrap the `permit()` function in a try-catch clause. Continue transaction if `permit()` succeeds. If `permit()` fails, check that the allowance is sufficient and continue anyway. Another potential fix is to introduce the same `didPrepay` bool into `repay` as was done in `deposit`. If not prepaid, use direct `safeTransferFrom` method instead.

Team Response

Fixed.

[M-07] Lack of `deadline` Check in the `BadDebtProcessor::uniswapV3FlashCallback()` Could Lead To Loss of Funds to the User

Severity

Medium Risk

Description

The `UNISWAP_POOL().swap()` is performed in the `FrontendManager::callback()` when the `action == 6` and in the `BoostManager::_action2Burn()` functions. In both those occasions, the `deadline` check is performed in addition to the slippage checks. This is to ensure that the transaction is not executed after a delay at an unfavourable price point to the user. The `slippage` check alone can not mitigate this issue since slippage applies to the derived swap value at the time of the transaction execution.

But the `UNISWAP_POOL().swap()` performed in the `BadDebtProcessor::uniswapV3FlashCallback()` only performs the slippage check but does not perform a deadline check.

Location of Affected Code

File: src/BadDebtProcessor.sol

```
if (exactAmountOut < 0) {
// Need to swap `token1` (the one we received during liquidation) for `
    token0` (which is `flashToken`)
    (, swapped) = borrower.UNISWAP_POOL().swap(
        address(this), false, exactAmountOut, TickMath.MAX_SQRT_RATIO -
            1, bytes("")
    );
    require(uint256(swapped) < recovered * slippage / 10_000, "slippage")
        ;
}
```

Impact

The `BadDebtProcessor::process()` transaction can be executed after a significant delay, which could prompt the asset ratios in the UniswapV3Pool to have changed considerably.

Hence the swapped and recovered asset amounts are calculated based on the highly altered price points. As a result, the provided slippage is stale and will not provide the required protection to the user, thus incurring loss to the user.

Recommendation

Hence it is recommended to implement the deadline check in the `BadDebtProcessor::uniswapV3FlashCallback()` function to ensure execution of `BadDebtProcessor::proce` is not allowed after the given deadline.

The `deadline` should be passed as a user input to the `BadDebtProcessor::process()` function. This will ensure users will not lose funds due to delayed execution with stale slippage checks.

Team Response

Fixed.

[M-08] Unfair Liquidation of Healthy Accounts

Severity

Medium Risk

Description

A borrower can be unfairly liquidated even after becoming healthy if borrowing is paused and modification is restricted, and a liquidator front-runs the borrower's `Borrower::clear()` transaction. The `Borrower.liquidate()` function does not check if the borrower is currently healthy `BalanceSheet.isHealthy()`, relying solely on the `warnTime` being greater than zero.

Furthermore, the `Borrower::clear()` can be called after the `LIQUIDATION_GRACE_PERIOD` to make an account healthy again. This allows a depositor to make their account healthy again by depositing directly to the lender and replenishing their ante even after the `LIQUIDATION_GRACE_PERIOD` has passed.

But the issue here is that a liquidator who doesn't immediately liquidate an unhealthy account (until 7 minutes have passed) since there is no incentive could observe this `Borrower::clear()` transaction and decide to front-run it with `liquidate()` transaction setting the `CloseFactor == 10000`. This will enable the liquidator to liquidate a healthy position maliciously and gain the ante as an incentive.

The vulnerability occurs in the following scenario:

1. A borrower's account becomes unhealthy and is warned via `Borrower.warn()`.
2. Borrowing is `paused` via `Factory.pause()`. This prevents the borrower from using `Borrower.modify()`.
3. The `LIQUIDATION_GRACE_PERIOD` expires.
4. The borrower decides to make his account healthy by directly depositing to the `Lender` contract and replenishing the ante in the `Borrower` contract. This would make the `BalanceSheet.isHealthy()` true again.
5. The borrower attempts to call `Borrower.clear()` to remove the warning status.
6. A liquidator observes the pending clear transaction and `front-runs` it with a `Borrower.liquidate()` transaction with `CloseFactor == 10000`.
7. Because `Borrower.liquidate()` only checks `warnTime > 0` and not `BalanceSheet.isHealthy()`, the liquidation proceeds.
8. The liquidator receives the borrower's full ante amount as a liquidation incentive, even though the borrower's account was healthy at the time of liquidation.

Location of Affected Code

File: src/Borrower.sol

```
function liquidate(ILiquidator callee, bytes calldata data, uint256
    closeFactor, uint40 oracleSeed) external {
     require(0 < closeFactor && closeFactor <= 10000, "Stacking Salmon:
        close");

     uint256 slot0_ = slot0;
     require(slot0_ & SLOT0_MASK_STATE == 0 && slot0_ & SLOT0_MASK_AUCTION
         > 0);
     slot0 = slot0_ | (uint256(State.Locked) << 248);

     _uniswapWithdraw(slot0_);

     (Prices memory prices, , , ) = getPrices(oracleSeed);

     (uint256 assets0, uint256 assets1) = (TOKEN0.balanceOf(address(this))
        , TOKEN1.balanceOf(address(this)));

     (uint256 liabilities0, uint256 liabilities1) = getLiabilities();
```

```
    {
        (uint160 sqrtPriceX96, , , , , , ) = UNISWAP_POOL.slot0();
        require(prices.a < sqrtPriceX96 && sqrtPriceX96 < prices.b);
    }

    AuctionAmounts memory amounts = BalanceSheet.computeAuctionAmounts(
        prices.c,
        assets0,
        assets1,
        liabilities0,
        liabilities1,
        BalanceSheet.auctionTime((slot0_ & SLOT0_MASK_AUCTION) >> 208),
        closeFactor
    );

    if (amounts.out0 > 0) TOKEN0.safeTransfer(address(callee), amounts.
       out0);
    if (amounts.out1 > 0) TOKEN1.safeTransfer(address(callee), amounts.
       out1);

    callee.callback(data, msg.sender, amounts);

    if (amounts.repay0 > 0) LENDER0.repay(amounts.repay0, address(this));
    if (amounts.repay1 > 0) LENDER1.repay(amounts.repay1, address(this));

    if (closeFactor == 10000) {
        SafeTransferLib.safeTransferETH(payable(callee), address(this).
           balance);
        slot0_ &= ~SLOT0_MASK_AUCTION;
    } else if (closeFactor > TERMINATING_CLOSE_FACTOR) {
        if (
            BalanceSheet.isHealthy(
                prices,
                assets0 - amounts.out0,
                assets1 - amounts.out1,
                liabilities0 - amounts.repay0,
                liabilities1 - amounts.repay1
            )
        ) slot0_ &= ~SLOT0_MASK_AUCTION;
    }

    slot0 = (slot0_ & (SLOT0_MASK_USERSPACE | SLOT0_MASK_AUCTION)) |
       SLOT0_DIRT;
    emit Liquidate();
}
```

Impact

- **Unfair Liquidation**: Borrowers who have taken steps to become healthy can be liquidated, losing their ante.
- **Liquidators making unfair profit**: Liquidators can profit by liquidating healthy accounts, ex-

ploiting a race condition.

- **Loss of Funds**: The borrower loses funds (up to the full ante value) unfairly.

Recommendation

Implement an `isHealthy()` check at the beginning of the `Borrower.liquidate()` function, before setting the Borrower's state to `locked`. This will prevent the liquidation of accounts that are currently healthy.

Team Response

Fixed.

[L-01] Signatures Have No Deadline and Cannot Be Cancelled

Severity

Low Risk

Description

The lack of a deadline or inability to be cancelled exists in the following contracts: - `UniswapNFTManager.sol` 's `callback()` allows using signature permit withdrawal from the NFT. The permit is, however, flawed in that it has no deadline, and cannot be cancelled. - `Lender.sol` 's `permit` signature cannot be cancelled too but has a deadline The signatures cannot be cancelled because the owner's `nonce` is increased in code, it cannot be directly increased by the owner if he wishes to, therefore cannot be manually invalidated. - `BoostManager::_action0Mint()` function due to lack of a deadline check on the signature

Location of Affected Code

File: UniswapNFTManager.sol

```
require(
    owner == UNISWAP_NFT.ownerOf(tokenId) &&
        SignatureCheckerLib.isValidSignatureNowCalldata(
            owner,
             keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    DOMAIN_SEPARATOR(),
                    keccak256(
                        abi.encode(
                            keccak256(
                                "Permit(address owner,address spender,
                                   uint128 liquidity,uint256 nonce,
                                   uint256 tokenId)"
                            ),
                            owner,
                            msg.sender,
                            uint128(-liquidity),
                            nonces[owner]++,
                            tokenId
                        )
                    )
                )
            ),
            data[660:]
        )
);
```

File: Lender.sol

```
address recoveredAddress = ecrecover(
    keccak256(
        abi.encodePacked(
            "\x19\x01",
            DOMAIN_SEPARATOR(),
            keccak256(
                abi.encode(
                    keccak256(
                        "Permit(address owner,address spender,uint256
                           value,uint256 nonce,uint256 deadline)"
                    ),
                    owner,
                    spender,
                    value,
                    nonces[owner]++,
                    deadline
                )
            )
        )
    ),
    v,
    r,
    s
);
```

File: BoostManager.sol

```
require(
    owner == UNISWAP_NFT.ownerOf(tokenId) &&
        SignatureCheckerLib.isValidSignatureNow(
            owner,
            keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    DOMAIN_SEPARATOR(),
                    keccak256(
                        abi.encode(
                            keccak256(
                                "Permit(address owner,address spender,
                                   uint128 liquidity,uint256 nonce,
                                   uint256 tokenId)"
                            ),
                            owner,
                            msg.sender,
                            liquidity,
                            nonces[owner]++,
                            tokenId
                        )
                    )
                )
            ),
            signature
        )
);
```

Impact

The lack of a way to increase nonce puts owners at risk if permission is granted to the wrong address, as such signatures cannot be cancelled. The lack of a deadline means the signature never expires and can be used long after it has been granted.

Recommendation

Consider applying the following changes:

- Introduce a `deadline` parameter in `UniswapNFTManager.sol`'s `permit()` and `_action0Mint()` in the `BoostManager`.
- Introduce a `public` function like Open Zeppelin's useNonce that will increase a user's nonce when it's called, thereby cancelling the signatures.

Team Response

Fixed.

[L-02] Potential Owner/Operator Backdoor in ERC721z Tokens

Severity

Low Risk

Description

Token owners or operators can approve themselves to a particular token ID.

Location of Affected Code

File: ERC721Z.sol

```
function approve(address spender, uint256 tokenId) public virtual {
    address owner = ownerOf(tokenId);

    require(msg.sender == owner || isApprovedForAll[owner][msg.sender], "
       NOT_AUTHORIZED");

    getApproved[tokenId] = spender;

    emit Approval(owner, spender, tokenId);
}
```

Impact

If the `isApprovedForAll()` status of the user is removed, his self-approval to the `tokenId` via `approve()` still stands, providing a potential approval backdoor.

Recommendation

Add a check to ensure that owners or users approved for all cannot approve themselves to individual token IDs.

Team Response

Fixed.

[L-03] Lender Share Should Not Query Allowance from Owners

Severity

Low Risk

Description

In `Lender.sol`'s `transferFrom()`, allowance is only skipped if the sender is granted max approval. As a result, even if the owner attempts to use the `transferFrom()` method on his own tokens, the allowance will still be checked.

Location of Affected Code

File: Lender.sol#L321

```
function transferFrom(address from, address to, uint256 shares) external
    returns (bool) {
     uint256 allowed = allowance[from][msg.sender];
     if (allowed != type(uint256).max) allowance[from][msg.sender] =
         allowed - shares;
     accrueInterest();
     _transfer(from, to, shares);
     return true;
}
```

Impact

Allowance is checked and consumed even if the sender is the owner.

Recommendation

If the `from` address is equal to the `msg.sender`, skip the allowance check.

Team Response

Fixed.

[L-04] Missing Check for `tokenId` Existence Before Returning the `tokenURI`

Severity

Low Risk

Description

The `tokenURI` in `BorrowerNFT.sol` overrides ERC721Z's implementation. The implementation does not check that `tokenId` exists and hence will return data for non-existent `tokenId`. This goes contrary to EIP721 which states:

```
> /// @notice A distinct Uniform Resource Identifier (URI) for a given
    asset.
> /// @dev Throws if `_tokenId` is not a valid NFT. URIs are defined in
    RFC
> ///  3986. The URI may point to a JSON file that conforms to the "
    ERC721
> ///  Metadata JSON Schema".
> function tokenURI(uint256 _tokenId) external view returns (string);
```

Location of Affected Code

File: src/borrower-nft/BorrowerNFT.sol

```
function tokenURI(uint256 tokenId) external view override returns (string
    memory) {
    return URI_SOURCE.uriOf(_borrowerOf(tokenId));
}
```

Impact

Lack of compliance with the EIP-721 standard and integration issues with NFT marketplaces.

Recommendation

Consider sticking to the EIP-721 standard.

Team Response

Fixed.

[L-05] Usage of Transfer Instead of Call For Native Funds

Severity

Low Risk

Description

The contract uses Solidity's `transfer()` when transferring native tokens in `BadDebtProcessor.sol` and `Liquidator.sol`. This has some notable shortcomings which can render ETH impossible to transfer. The transfer will inevitably fail if: * The gas cost of EVM instructions changes significantly during hard forks (transfer opcode sends a fixed 2300 gas) * The recipient is a smart contract which does not implement a payable fallback function * Implements a `payable fallback` function which would incur more than 2300 gas units * Implements a `payable fallback` function incurring less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300

Location of Affected Code

File: src/BadDebtProcessor.sol

```
>   if (closeFactor == 10000) payable(withdrawFrom).transfer(address(this)
    .balance);
```

File: src/Liquidator.sol

```
function liquidate(Borrower borrower, bytes calldata data, uint256
    closeFactor, uint40 oracleSeed) external {
     borrower.liquidate(this, data, closeFactor, oracleSeed);
     if (closeFactor == 10000) payable(msg.sender).transfer(address(this).
        balance);
}
```

Impact

Inability to receive ETH dossing `uniswapV3FlashCallback()` in BadDebtProcessor.sol and `liquidate()` in `Liquidator.sol`

Recommendation

Use the low-level `call.value(amount)` with the corresponding result check or use the OpenZeppelin `Address.sendValue()`.

It is also recommended to follow the CEI pattern or add a reentrancy guard to the functions as the call opcode can leave a contract vulnerable to reentrancy attacks.

Team Response

Fixed.

[L-06] Unnecessary Allowance Buildup in the `UniswapNFTManager`

Severity

Low Risk

Description

In `UniswapNFTManager.sol`, `callback()` approves Uniswap NFT to spend amount to burn + 1 by only sending amount to burn as amount desired in the Uniswap NFT increase liquidity calls. This could lead to significant consequences.

Location of Affected Code

File: src/managers/UniswapNFTManager.sol

```
ERC20 token0 = borrower.TOKEN0();
ERC20 token1 = borrower.TOKEN1();

(uint256 burned0, uint256 burned1, , ) = borrower.uniswapWithdraw(
    lower,
    upper,
    uint128(liquidity),
    address(this)
);

>1 token0.safeApprove(address(UNISWAP_NFT), burned0 + 1);
>2 token1.safeApprove(address(UNISWAP_NFT), burned1 + 1);
UNISWAP_NFT.increaseLiquidity(
    IUniswapPositionNFT.IncreaseLiquidityParams({
        tokenId: tokenId,
>3      amount0Desired: burned0,
>4      amount1Desired: burned1,
        amount0Min: 0,
        amount1Min: 0,
        deadline: block.timestamp
    })
);

token0.safeTransfer(owner, token0.balanceOf(address(this)));
token1.safeTransfer(owner, token1.balanceOf(address(this)));
```

Impact

The contract has no other way to approve tokens, as a result, there will be a build-up of allowance to Uniswap NFTs in the contract. This means if the NFT gets compromised, the allowance buildup will make it possible to siphon any available tokens from the manager.

Also, with certain tokens (tokens with approval race protection), the extra allowance will cause permanent DOS as the next approval call will require the previous allowance to be 0, which it will not be in this case. See USDT on Ethereum mainnet as an example. Berachain is a relatively new chain, so newly listed tokens may include this design.

Recommendation

Zero out approvals after operations to be safe, or approve only the needed amount.

Team Response

Fixed.

[L-07] The Permit Functionality in `Lender` Could Not Operate with Smart Contract Multisig Wallets

Severity

Low Risk

Description

The `Lender::permit()` function only supports `EOA Signature Verification` and does not support `EIP1271`. Hence `Lender::permit()` is not compatible with `smart contract multisig wallets` such as `Gnosis Safe`.

But the `Staking Salmon protocol` supports `EIP1271` in other instances such as in the `BoostManager::_actionOMint()` which calls the `SignatureCheckerLib.isValidSignatureNow()` which supports both the `EOA signature verification and EIP1271`.

Location of Affected Code

File: src/Lender.sol

```
// Unchecked because the only math done is incrementing
// the owner's nonce which cannot realistically overflow.
unchecked {
  address recoveredAddress = ecrecover(
      keccak256(
        abi.encodePacked(
            "\x19\x01",
            DOMAIN_SEPARATOR(),
            keccak256(
                abi.encode(
                    keccak256(
                        "Permit(address owner,address spender,uint256
                           value,uint256 nonce,uint256 deadline)"
                    ),
                    owner,
                    spender,
                    value,
                    nonces[owner]++,
                    deadline
                )
            )
        )
    ),
    v,
    r,
    s
  );

  require(recoveredAddress != address(0) && recoveredAddress == owner, "
     Stacking Salmon: permit invalid");

  allowance[recoveredAddress][spender] = value;
}
```

Impact

The `Router::depositWithPermit2()` function calls the `Lender::permit()` function to initially set the allowance to `type(uint256).max` and then calls the `Lender::deposit()` after transferring the assets to the `Lender` contract.

Therefore, the `Router::depositWithPermit2()` functionality will `DoS` for the smart contract multi-sig wallets since the `Lender::permit()` does not support the `EIP1271`.

Recommendation

Hence it is recommended to implement the `EIP1271` smart contract signature verification in the `Lender::permit()` function to enable the smart contract multisig wallets to work with the `Router::depositWithPermit2()` function.

Team Response

Fixed.

[L-08] Usage of `sqrtPriceX96` Spot Price Is Susceptible to Price Manipulation

Severity

Low Risk

Description

The `_action0Mint()` function in `BoostManager` calls the `getAmountsForLiquidity()` in `LiquidityAmounts`, to convert the leveraged Liquidity amount to the required token amounts as shown below:

```
(uint160 sqrtPriceX96, , , , , , ) = borrower.UNISWAP_POOL().slot0();
(uint256 needs0, uint256 needs1) = LiquidityAmounts.
   getAmountsForLiquidity(
    sqrtPriceX96,
    TickMath.getSqrtRatioAtTick(lower),
    TickMath.getSqrtRatioAtTick(upper),
    liquidity
);
```

The `sqrtPriceX96` used here is the `spot price` which is retrieved from `Pool.slot0`. Hence the `sqrtPriceX96` is highly `susceptible to price manipulation`.

Location of Affected Code

File: src/managers/BoostManager.sol

```
(uint160 sqrtPriceX96, , , , , , ) = borrower.UNISWAP_POOL().slot0();
(uint256 needs0, uint256 needs1) = LiquidityAmounts.
   getAmountsForLiquidity(
    sqrtPriceX96,
    TickMath.getSqrtRatioAtTick(lower),
    TickMath.getSqrtRatioAtTick(upper),
    liquidity
);
```

Impact

Hence the above issue could result in wrong derived amounts (needs0, needs1) during borrow amount calculations. As a result, the user could end up borrowing amounts of `token1 and token0` which is unfavourable to him. The `seemsLegit` flag derived in the `Borrower::getPrices()` function is used to identify UniswapV3 price manipulation and pause the borrowing accordingly.

But as per the Natspec given in the `Oracle.sol` contract, if the price is manipulated equally across the two adjacent time intervals then the metric will result in 0 hence `seemsLegit` will be true, thus making the `seemsLegit` check ineffective.

Recommendation

Hence it is recommended to use the `TWAP` price of the `UniswapV3Pool` for the `LiquidityAmounts.getAmountsForLiquidity` call to mitigate the spot price manipulation of UniswapV3Pool. If the protocol is willing to accept this risk, then it is recommended to document this risk factor, thus making the users aware of it.

Team Response

Acknowledged.

[L-09] Hardcoded and Potentially Vulnerable Slippage and Deadline Parameters

Severity

Low Risk

Description

In `UniswapNFTManager.sol` and `BoostManager.sol`, `increaseLiquidityParams()` and `decreaseLiquidityParams()` have their `minAmounts` hardcoded to 0, and the `deadline` is set as `block.timestamp`. This is dangerous, as it allows for sandwich attacks. The protocol has no control over how much is received from the transaction and how long the transaction spends in the mempool.

Location of Affected Code

File: src/managers/UniswapNFTManager.sol

```
function callback(bytes calldata data, address owner, uint208) external
    override returns (uint208 positions) {
// code
        UNISWAP_NFT.increaseLiquidity(
            IUniswapPositionNFT.IncreaseLiquidityParams({
                tokenId: tokenId,
                amount0Desired: burned0,
                amount1Desired: burned1,
                amount0Min: 0,
                amount1Min: 0,
                deadline: block.timestamp
            })
        );
}

function _withdrawFromNFT(uint256 tokenId, uint128 liquidity, address
    recipient) private {
        UNISWAP_NFT.decreaseLiquidity(
            IUniswapPositionNFT.DecreaseLiquidityParams({
                tokenId: tokenId,
                liquidity: liquidity,
                amount0Min: 0,
                amount1Min: 0,
                deadline: block.timestamp
            })
        );

// code
}
```

File: src/managers/BoostManager.sol

```
function _withdrawFromUniswapNFT( uint256 tokenId, uint128 liquidity,
    address recipient ) private returns (uint256 burned0, uint256 burned1)
     {
        IUniswapPositionNFT.DecreaseLiquidityParams({
            tokenId: tokenId,
            liquidity: liquidity,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        });

// code
}
```

Impact

These make the functions potentially vulnerable to slippage issues or sandwich attacks since the protocol has no control over how many tokens are returned from the call or how long the transaction takes.

Recommendation

Recommend using more reasonable parameters (e.g. a percentage of `amountIn`). An admin function can be created to always keep this updated.

Team Response

Fixed.

[L-10] The `isInUse()` Function Can Provide Wrong Information

Severity

Low Risk

Description

The `isInUse()` function queries the borrower's token balance and returns true if the balance is > 0. Depending on how the `isInUse()` function is used by external integrations, the result can easily be manipulated by any user by sending as little as 1 wei of `token0` or `token1` to the borrower.

Location of Affected Code

File: src/BorrowerLens.sol

```
function isInUse(Borrower borrower) external view returns (bool,
   IUniswapV3Pool) {
    IUniswapV3Pool pool = borrower.UNISWAP_POOL();

    if (borrower.getUniswapPositions().length > 0) return (true, pool);
>>  if (borrower.TOKEN0().balanceOf(address(borrower)) > 0) return (true,
    pool);
>>  if (borrower.TOKEN1().balanceOf(address(borrower)) > 0) return (true,
    pool);
    if (borrower.LENDER0().borrowBalanceStored(address(borrower)) > 0)
       return (true, pool);
    if (borrower.LENDER1().borrowBalanceStored(address(borrower)) > 0)
       return (true, pool);

    return (false, pool);
  }
```

Impact

A borrower that has no borrow balance stored or no Uniswap position can still be made to be seen as active by depositing at least 1 wei of any of the queried tokens.

Recommendation

Recommend not relying on `balanceOf()` for information, remove the checks or use a system of internal accounting instead.

Team Response

Acknowledged.

shieldify

Thank you!