SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Sentiment
Prepared by: Sherlock
Lead Security Expert: WATCHPUG
Dates Audited: August 29 - September 19, 2022
Prepared on: October 7, 2022

Introduction

Sentiment is a permissionless undercollateralised onchain credit protocol that allows users to lend and borrow assets with increased capital efficiency and deploy them across DeFi.

Scope

All contracts in the folders (that represent the repos above) except contracts in any `test` folder.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Total Issues

Medium	High
14	7

Security Experts

WATCHPUG
0x52
PwnPatrol
Lambda
GalloDaSballo
xiaoming90
hyh
lllllll
berndartmueller
Bahurum
JohnSmith
pashov
bytehat
Ruhum
cccz
GimelSec
devtooligan
csanuragjain
__141345__
kankodu
TomJ
CRYP70
sorrynotsorry
Kumpa
rbserver
Czar102
kirk-baird
panprog
HonorLt
bin2chen
hansfriese
0xc0ffEE
Tutturu
ellahi
Chom
carrot
defsec
icedpeachtea
oyc_109
0xNineDec
Avci
ladboy233
jonatascm
0xNazgul
0xf15ers
Dravee
Olivierdem

Issue H-1: A malicious early user/attacker can manipulate the LToken's pricePerShare to take an unfair share of future users' deposits

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/004-H

Found by

kankodu, JohnSmith, PwnPatrol, WATCHPUG, berndartmueller, hyh, __141345__, lllllll, TomJ

Summary

A well known attack vector for almost all shares based liquidity pool contracts, where an early user can manipulate the price per share and profit from late users' deposits because of the precision loss caused by the rather large value of price per share.

Vulnerability Detail

A malicious early user can `deposit()` with `1wei` of `asset` token as the first depositor of the LToken, and get `1wei` of shares.

Then the attacker can send `10000e18-1` of `asset` tokens and inflate the price per share from 1.0000 to an extreme value of 1.0000e22 (from `(1+10000e18-1)/1`) .

As a result, the future user who deposits `19999e18` will only receive `1wei` (from `19999e18*1/10000e18`) of shares token.

They will immediately lose `9999e18` or half of their deposits if they `redeem()` right after the `deposit()`.

Impact

The attacker can profit from future users' deposits. While the late users will lose part of their funds to the attacker.

Code Snippet

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/tokens/utils/ERC4626.sol#L48-L60

```
function deposit(uint256 assets, address receiver) public virtual returns
↪  (uint256 shares) {
    beforeDeposit(assets, shares);
```

```
    // Check for rounding error since we round down in previewDeposit.
    require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES");

    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), assets);

    _mint(receiver, shares);

    emit Deposit(msg.sender, receiver, assets, shares);
}
```

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/tokens/utils/ERC4626.sol#L138-L140

```
function previewDeposit(uint256 assets) public view virtual returns (uint256) {
    return convertToShares(assets);
}
```

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/tokens/utils/ERC4626.sol#L126-L131

```
function convertToShares(uint256 assets) public view virtual returns (uint256) {
    uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is
↪   non-zero.

    return supply == 0 ? assets : assets.mulDivDown(supply, totalAssets());
}
```

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/tokens/LToken.sol#L191-L193

```
function totalAssets() public view override returns (uint) {
    return asset.balanceOf(address(this)) + getBorrows() - getReserves();
}
```

Tool used

Manual Review

Recommendation

Consider requiring a minimal amount of share tokens to be minted for the first minter, and send a port of the initial mints as a reserve to the DAO so that the pricePerShare can be more resistant to manipulation.

```
function deposit(uint256 assets, address receiver) public virtual returns
↪  (uint256 shares) {
    beforeDeposit(assets, shares);

    // Check for rounding error since we round down in previewDeposit.
    require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES");

    // for the first mint, we require the mint amount > (10 ** decimals) / 100
    // and send (10 ** decimals) / 1_000_000 of the initial supply as a reserve
↪  to DAO
    if (totalSupply == 0 && decimals >= 6) {
        require(shares > 10 ** (decimals - 2));
        uint256 reserveShares = 10 ** (decimals - 6);
        _mint(DAO, reserveShares);
        shares -= reserveShares;
    }

    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), assets);

    _mint(receiver, shares);

    emit Deposit(msg.sender, receiver, assets, shares);
}

function mint(uint256 shares, address receiver) public virtual returns (uint256
↪  assets) {
    beforeDeposit(assets, shares);

    assets = previewMint(shares); // No need to check for rounding error,
↪  previewMint rounds up.

    // for the first mint, we require the mint amount > (10 ** decimals) / 100
    // and send (10 ** decimals) / 1_000_000 of the initial supply as a reserve
↪  to DAO
    if (totalSupply == 0 && decimals >= 6) {
        require(shares > 10 ** (decimals - 2));
        uint256 reserveShares = 10 ** (decimals - 6);
        _mint(DAO, reserveShares);
        shares -= reserveShares;
    }

    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), assets);

    _mint(receiver, shares);
```

```
    emit Deposit(msg.sender, receiver, assets, shares);
}
```

Issue H-2: `ChainlinkOracle.solgetPrice()` The price will be wrong when the token's USD price feed's `decimals!=8`

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/019-H

Found by

Lambda, csanuragjain, CRYP70, WATCHPUG, berndartmueller, pashov, lllllll, Bahurum, sorrynotsorry

Summary

`ChainlinkOracle` assumes and inexplicitly requires the token's USD feed's decimals to be 8. However, there are certain token's USD feed has a different `decimals`.

Vulnerability Detail

In the current implementation, it assumes $tokenFeedDecimals = ethFeedDecimals$ (`feed[token].decimals()` must equals `ethUsdPriceFeed.decimals()=8`).

However, there are tokens with USD price feed's decimals != 8 (E.g: AMPL/USD)

When the token's USD feed's decimals != 8, `ChainlinkOracle.solgetPrice()` will return an incorrect price in ETH.

The correct calculation formula should be:

$$\frac{answer_{token} \cdot 10^{ethFeedDecimals}}{answer_{eth} \cdot 10^{tokenFeedDecimals}} \cdot 10^{18}$$

PoC

Given:

- 1.0 AMPL worth 1.14 USD, `feed[ampl].decimals()==18`, `answer_ampl=1140608758261546000` Source: feed[ampl]
- 1.0 ETH worth 1588.11 USD, `ethUsdPriceFeed.decimals()==8`, `answer_eth=158811562094` Source: ethUsdPriceFeed

`chainlinkOracle.getPrice(AMPL)` will return ~7.18m (eth):

$$\frac{answer_{token} \cdot 10^{18}}{answer_{eth}} = \frac{1140608758261546000 \cdot 10^{18}}{158811562094} = 7182151873718260663354494$$

Impact

When the price feed with `decimals!=18` is set, the attacker can deposit a small amount of the asset and drain all the funds from the protocol.

Code Snippet

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/RiskEngine.sol#L178-L188

```
function _valueInWei(address token, uint amt)
    internal
    view
    returns (uint)
{
    return oracle.getPrice(token)
    .mulDivDown(
        amt,
        10 ** ((token == address(0)) ? 18 : IERC20(token).decimals())
    );
}
```

https://github.com/sentimentxyz/oracle/blob/59b26a3d8c295208437aad36c470386c9729a4bc/src/chainlink/ChainlinkOracle.sol#L47-L59

```
/// @inheritdoc IOracle
/// @dev feed[token].latestRoundData should return price scaled by 8 decimals
function getPrice(address token) external view virtual returns (uint) {
    (, int answer,,,) =
        feed[token].latestRoundData();

    if (answer < 0)
        revert Errors.NegativePrice(token, address(feed[token]));

    return (
        (uint(answer)*1e18)/getEthPrice()
    );
}
```

https://github.com/sentimentxyz/oracle/blob/59b26a3d8c295208437aad36c470386c9729a4bc/src/chainlink/ChainlinkOracle.sol#L65-L73

```
function getEthPrice() internal view returns (uint) {
    (, int answer,,,) =
        ethUsdPriceFeed.latestRoundData();

    if (answer < 0)
```

```
        revert Errors.NegativePrice(address(0), address(ethUsdPriceFeed));

    return uint(answer);
}
```

Tool used

Manual Review

Recommendation

Consider adding a check for `feed.decimals()` to make sure `feed`'s decimals = `8`:

```
constructor(AggregatorV3Interface _ethUsdPriceFeed) Ownable(msg.sender) {
    require(_ethUsdPriceFeed.decimals() == 8, "...");
    ethUsdPriceFeed = _ethUsdPriceFeed;
}
```

```
function setFeed(
    address token,
    AggregatorV3Interface _feed
) external adminOnly {
    require(_feed.decimals() == 8, "...");
    feed[token] = _feed;
    emit UpdateFeed(token, address(_feed));
}
```

Issue H-3: CTokenOracle.sol#getCErc20Price contains critical math error

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/021-H

Found by

0x52

Summary

CTokenOracle.sol#getCErc20Price contains a math error that immensely overvalues CTokens

Vulnerability Detail

CTokenOracle.sol#L66-L76

```
function getCErc20Price(ICToken cToken, address underlying) internal view returns
↪  (uint) {
    /*
        cToken Exchange rates are scaled by 10^(18 - 8 + underlying token
        ↪  decimals) so to scale
        the exchange rate to 18 decimals we must multiply it by 1e8 and then
        ↪  divide it by the
        number of decimals in the underlying token. Finally to find the price of
        ↪  the cToken we
        must multiply this value with the current price of the underlying token
    */
    return cToken.exchangeRateStored()
    .mulDivDown(1e8 , IERC20(underlying).decimals())
    .mulWadDown(oracle.getPrice(underlying));
}
```

In L74, IERC20(underlying).decimals() is not raised to the power of 10. The results in the price of the LP being overvalued by many order of magnitudes. A user could deposit one CToken and drain the reserves of every liquidity pool.

Impact

All lenders could be drained of all their funds due to excessive over valuation of CTokens cause by this error

Code Snippet

CTokenOracle.sol#L66-L76

Tool used

Manual Review

Recommendation

Fix the math error by changing L74:

```
return cToken.exchangeRateStored()
.mulDivDown(1e8 , 10 ** IERC20(underlying).decimals())
.mulWadDown(oracle.getPrice(underlying));
```

Issue H-4: `ERC4626Oracle` Price will be wrong when the ERC4626's `decimals` is different from the underlying token's decimals

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/025-H

Found by

Lambda, JohnSmith, WATCHPUG, 0x52, berndartmueller, Bahurum

Summary

EIP-4626 does not require the decimals must be the same as the underlying tokens' decimals, and when it's not, `ERC4626Oracle` will malfunction.

Vulnerability Detail

In the current implementation, `IERC4626(token).decimals()` is used as the `IERC4626(token).asset()`'s decimals to calculate the ERC4626's price.

However, while most ERC4626s are using the underlying token's decimals as `decimals`, there are some ERC4626s use a different decimals from underlying token's decimals since EIP-4626 does not require the decimals must be the same as the underlying token's decimals:

> Although the convertTo functions should eliminate the need for any use of an EIP-4626 Vault's decimals variable, it is still strongly recommended to mirror the underlying token's decimals if at all possible, to eliminate possible sources of confusion and simplify integration across front-ends and for other off-chain users.

Ref: https://eips.ethereum.org/EIPS/eip-4626

Impact

The price of ERC4626 will be significantly underestimated when the underlying token's decimals > ERC4626's decimals, and be significantly overestimated when the underlying token's decimals < ERC4626's decimals.

Code Snippet

https://github.com/sentimentxyz/oracle/blob/59b26a3d8c295208437aad36c470386c9729a4bc/src/erc4626/ERC4626Oracle.sol#L35-L43

```
function getPrice(address token) external view returns (uint) {
    uint decimals = IERC4626(token).decimals();
    return IERC4626(token).previewRedeem(
        10 ** decimals
    ).mulDivDown(
        oracleFacade.getPrice(IERC4626(token).asset()),
        10 ** decimals
    );
}
```

Tool used

Manual Review

Recommendation

`getPrice()` can be changed to:

```
function getPrice(address token) external view returns (uint) {
    uint decimals = IERC4626(token).decimals();
    address underlyingToken = IERC4626(token).asset();
    return IERC4626(token).previewRedeem(
        10 ** decimals
    ).mulDivDown(
        oracleFacade.getPrice(underlyingToken),
        10 ** IERC20Metadata(underlyingToken).decimals()
    );
}
```

Issue H-5: `UniV2LPOracle` will malfunction if token0 or token1's `decimals!=18`

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/026-H

Found by

Lambda, WATCHPUG, 0x52, hyh

Summary

When one of the LP token's underlying tokens `decimals` is not 18, the price of the LP token calculated by `UniV2LPOracle` will be wrong.

Vulnerability Detail

`UniV2LPOracle` is an implementation of Alpha Homora v2's Fair Uniswap's LP Token Pricing Formula:

> The Formula ... of combining fair asset prices and fair asset reserves:
>
> $$P = 2 \cdot \frac{\sqrt{r_0 \cdot r_1} \cdot \sqrt{p_0 \cdot p_1}}{totalSupply},$$
>
> where r_i is the asset ii's pool balance and p_i is the asset i's fair price.

However, the current implementation wrongful assumes r_0 and r_1 are always in 18 decimals.

https://github.com/sentimentxyz/oracle/blob/59b26a3d8c295208437aad36c470386c9729a4bc/src/uniswap/UniV2LPOracle.sol#L39-L50

```
function getPrice(address pair) external view returns (uint) {
    (uint r0, uint r1,) = IUniswapV2Pair(pair).getReserves();

    // 2 * sqrt(r0 * r1 * p0 * p1) / totalSupply
    return FixedPointMathLib.sqrt(
        r0
        .mulWadDown(r1)
        .mulWadDown(oracle.getPrice(IUniswapV2Pair(pair).token0()))
        .mulWadDown(oracle.getPrice(IUniswapV2Pair(pair).token1()))
    )
    .mulDivDown(2e27, IUniswapV2Pair(pair).totalSupply());
}
```

https://github.com/transmissions11/solmate/blob/main/src/utils/FixedPointMathLib.sol

```
uint256 internal constant WAD = 1e18; // The scalar of ETH and most ERC20s.

function mulWadDown(uint256 x, uint256 y) internal pure returns (uint256) {
    return mulDivDown(x, y, WAD); // Equivalent to (x * y) / WAD rounded down.
}
```

https://github.com/transmissions11/solmate/blob/main/src/utils/FixedPointMathLib.sol

```
function mulDivDown(
    uint256 x,
    uint256 y,
    uint256 denominator
) internal pure returns (uint256 z) {
    assembly {
        // Store x * y in z for now.
        z := mul(x, y)

        // Equivalent to require(denominator != 0 && (x == 0 || (x * y) / x ==
↪   y))
        if iszero(and(iszero(iszero(denominator)), or(iszero(x), eq(div(z, x),
↪   y)))) {
            revert(0, 0)
        }

        // Divide z by the denominator.
        z := div(z, denominator)
    }
}
```

Impact

When the decimals of one or both tokens in the pair is not 18, the price will be way off.

Code Snippet

We've created a test script to demonstrate `UniV2LPOracle` is malfunctioning with USDC/WETH, in which USDC's decimals is 6 instead of 18.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;
```

```
import "forge-std/console.sol";
import {Test} from "forge-std/Test.sol";

import {IOracle} from "../core/IOracle.sol";
import {UniV2LpOracle} from "../uniswap/UniV2LPOracle.sol";

import {OracleFacade} from "../core/OracleFacade.sol";
import {ChainlinkOracle} from "../chainlink/ChainlinkOracle.sol";
import {WETHOracle} from "../weth/WETHOracle.sol";
import {AggregatorV3Interface} from "../chainlink/AggregatorV3Interface.sol";
import "forge-std/console.sol";

contract UniV2LPOracleTest is Test {
    address constant usdc = address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    address constant weth = address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);

    address constant pair = address(0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc);

    address constant ethUsdFeed =
        address(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419);
    address constant usdcUsdFeed =
        address(0x8fFfFfd4AfB6115b954Bd326cbe7B4BA576818f6);

    ChainlinkOracle chainlinkOracle;
    WETHOracle wETHOracle;
    UniV2LpOracle uniV2LpOracle;
    OracleFacade oracleFacade;

    function setUp() public {
        // core oracle
        oracleFacade = new OracleFacade();

        chainlinkOracle = new ChainlinkOracle(
            AggregatorV3Interface(ethUsdFeed)
        );
        chainlinkOracle.setFeed(usdc, AggregatorV3Interface(usdcUsdFeed));

        WETHOracle wethOracle = new WETHOracle();
        oracleFacade.setOracle(weth, wethOracle);
        oracleFacade.setOracle(usdc, chainlinkOracle);

        uniV2LpOracle = new UniV2LpOracle(oracleFacade);
    }

    function testUniV2Price() public {
        console.log(uniV2LpOracle.getPrice(pair));
    }
}
```

`reserves` **and** `totalSupply` **of UniswapV2 USDC/ETH Pair** `0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc` **on Mainnet at the time of writing** Result of `getReserves()`

```
_reserve0   uint112 :  45456843739761
_reserve1   uint112 :  28342500764440756425363
_blockTimestampLast   uint32 :  1663233599
```

`totalSupply()`: 550760227054391377

Expected and actual result

```
// real time price
2834250076444075642536 3 * 2   / 550760227054391377 = 102921

// normalized r0 and r1
102728696484607347546879 / 1e18 = 102728

// current result
102728772378134052 / 1e18 = 0.10272877237813405
```

Tool used

Manual Review

Recommendation

Consider normalizing r0 and r1 to 18 decimals before using them in the formula.

Issue H-6: `updateState()` should be called in `depositEth()` and `redeemEth()`

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/085-H

Found by

Lambda, Ruhum, bytehat, WATCHPUG, xiaoming90, pashov, cccz, GimelSec

Summary

Whenever the liquidity of a LToken changes, `getRateFactor` will be changed.

Therefore, `updateState()` must be called prior to the change to settle the pending interests.

Vulnerability Detail

Alice is a liquidity provider for LEther, Bob is a borrower.

1. Alice added 1000 ETH;
2. Bob borrowed 500 ETH; In which `updateState()` is called. The `util` in `getBorrowRatePerSecond` is `0.5` now.
3. One year later (no one interacts with the `asset` for 1 year), Alice redeemed 500 ETH with `redeemEth()`, in which `updateState()` is not called. The `util` in `getBorrowRatePerSecond` is `1` now.
4. Bob called `repay()`, in which `updateState()` is called to calculates the pending interest:

For Bob the borrower, the sum of principal and interest is:

$$BorrowRatePerSecond = c3 \cdot (util \cdot c1 + util^{32} \cdot c1 + util^{64} \cdot c2) \div secsPerYear = 5545529241$$

$$rateFactor = BorrowRatePerSecond \cdot secsPerYear \div 1e18 = 175000000081490720$$

$$sum = borrow \cdot rateFactor \div 1e18 + borrow = 587$$

But the actual sum is as below, due to `updateState()` is not called in `redeemEth()`:

$$BorrowRatePerSecond = c3 \cdot (util \cdot c1 + util^{32} \cdot c1 + util^{64} \cdot c2) \div secsPerYear = 55455292386$$

$$rateFactor = BorrowRatePerSecond \cdot secsPerYear \div 1e18 = 1750000000000000000$$

$$sum = borrow \cdot rateFactor \div 1e18 + borrow = 1375$$

As a result, Bob the borrower is now paying `1375` instead of `587` for the interest, which is 2x the expected amount.

On the other hand, if another liquidity provider called `depositEth()` before Bob repays the loan, the actual interest can be lower than expected, which constitutes a loss of yields to Alice.

Impact

Incorrect amounts of interests will be paid by the borrowers, which can result in loss of yields to the lenders or overpaid interest for the borrowers.

Code Snippet

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/tokens/LToken.sol#L200-L227

```
function updateState() public {
    if (lastUpdated == block.timestamp) return;
    uint rateFactor = getRateFactor();
    uint interestAccrued = borrows.mulWadUp(rateFactor);
    borrows += interestAccrued;
    reserves += interestAccrued.mulWadUp(reserveFactor);
    lastUpdated = block.timestamp;
}

/* ------------------------------------------------------------------------ */
/*                            INTERNAL FUNCTIONS                            */
/* ------------------------------------------------------------------------ */

/**
    @dev Rate Factor = Timestamp Delta * 1e18 (Scales timestamp delta to 18
 ↪  decimals) * Interest Rate Per Block
        Timestamp Delta = Number of seconds since last update
*/
function getRateFactor() internal view returns (uint) {
```

```
    return (block.timestamp == lastUpdated) ?
        0 :
        ((block.timestamp - lastUpdated)*1e18)
        .mulWadUp(
            rateModel.getBorrowRatePerSecond(
                asset.balanceOf(address(this)),
                borrows
            )
        );
}
```

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/DefaultRateModel.sol#L51-L77

```
function getBorrowRatePerSecond(
        uint liquidity,
        uint borrows
    )
        external
        view
        returns (uint)
    {
        uint util = _utilization(liquidity, borrows);
        return c3.mulDivDown(
            (
                util.mulWadDown(c1)
                + util.rpow(32, SCALE).mulWadDown(c1)
                + util.rpow(64, SCALE).mulWadDown(c2)
            ),
            secsPerYear
        );
    }

    function _utilization(uint liquidity, uint borrows)
        internal
        pure
        returns (uint)
    {
        uint totalAssets = liquidity + borrows;
        return (totalAssets == 0) ? 0 : borrows.divWadDown(totalAssets);
    }
```

`updatestate` must be called everytime balance of asset is changed. https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/tokens/LEther.sol#L26-L53

```
/**
```

```
    @notice Wraps Eth sent by the user and deposits into the LP
        Transfers shares to the user denoting the amount of Eth deposited
    @dev Emits Deposit(caller, owner, assets, shares)
*/
function depositEth() external payable {
    uint assets = msg.value;
    uint shares = previewDeposit(assets);
    require(shares != 0, "ZERO_SHARES");
    IWETH(address(asset)).deposit{value: assets}();
    _mint(msg.sender, shares);
    emit Deposit(msg.sender, msg.sender, assets, shares);
}

/**
    @notice Unwraps Eth and transfers it to the caller
        Amount of Eth transferred will be the total underlying assets that
        are represented by the shares
    @dev Emits Withdraw(caller, receiver, owner, assets, shares);
    @param shares Amount of shares to redeem
*/
function redeemEth(uint shares) external {
    uint assets = previewRedeem(shares);
    _burn(msg.sender, shares);
    emit Withdraw(msg.sender, msg.sender, msg.sender, assets, shares);
    IWETH(address(asset)).withdraw(assets);
    msg.sender.safeTransferEth(assets);
}
```

Tool used

Manual Review

Recommendation

`beforeDeposit()` should be called in `depositEth()` and `redeemEth()`:

```
/**
    @notice Wraps Eth sent by the user and deposits into the LP
        Transfers shares to the user denoting the amount of Eth deposited
    @dev Emits Deposit(caller, owner, assets, shares)
*/
function depositEth() external payable {
    uint assets = msg.value;
    uint shares = previewDeposit(assets);
    require(shares != 0, "ZERO_SHARES");
    beforeDeposit(assets, shares);
    IWETH(address(asset)).deposit{value: assets}();
```

```
    _mint(msg.sender, shares);
    emit Deposit(msg.sender, msg.sender, assets, shares);
}

/**
    @notice Unwraps Eth and transfers it to the caller
        Amount of Eth transferred will be the total underlying assets that
        are represented by the shares
    @dev Emits Withdraw(caller, receiver, owner, assets, shares);
    @param shares Amount of shares to redeem
*/
function redeemEth(uint shares) external {
    uint assets = previewRedeem(shares);
    beforeWithdraw(assets, shares);
    _burn(msg.sender, shares);
    emit Withdraw(msg.sender, msg.sender, msg.sender, assets, shares);
    IWETH(address(asset)).withdraw(assets);
    msg.sender.safeTransferEth(assets);
}
```


Issue H-7: Tokens received from Curve's `remove_liquidity()` should be added to the assets list even if `_min_amounts` are set to `0`

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/267-H

Found by

WATCHPUG

Summary

Curve controller's `canRemoveLiquidity()` should return all the underlying tokens as `tokensIn` rather than only the tokens with `minAmount>0`.

Vulnerability Detail

https://github.com/sentimentxyz/controller/blob/a2ddbcc00f361f733352d9c51457b4ebb999c8ae/src/curve/StableSwap2PoolController.sol#L129-L152

```
function canRemoveLiquidity(address target, bytes calldata data)
    internal
    view
    returns (bool, address[] memory, address[] memory)
{
    (,uint256[2] memory amounts) = abi.decode(
        data[4:],
        (uint256, uint256[2])
    );

    address[] memory tokensOut = new address[](1);
    tokensOut[0] = target;

    uint i; uint j;
    address[] memory tokensIn = new address[](2);
    while(i < 2) {
        if(amounts[i] > 0)
            tokensIn[j++] = IStableSwapPool(target).coins(i);
        unchecked { ++i; }
    }
    assembly { mstore(tokensIn, j) }

    return (true, tokensIn, tokensOut);
}
```

The `amounts` in Curve controller's `canRemoveLiquidity()` represent the "Minimum amounts of underlying coins to receive", which is used for slippage control.

At L144-149, only the tokens that specified a minAmount > 0 will be added to the `tokensIn` list, which will later be added to the account's assets list.

We believe this is wrong as regardless of the minAmount `remove_liquidity()` will always receive all the underlying tokens.

Therefore, it should not check and only add the token when it's minAmount > 0.

Impact

When the user set `_min_amounts = 0` while removing liquidity from `Curve` and the withdrawn tokens are not in the account's assets list already, the user may get liquidated sooner than expected as `RiskEngine.sol_getBalance()` only counts in the assets in the assets list.

Code Snippet

https://arbiscan.io/address/0x7f90122bf0700f9e7e1f688fe926940e8839f353#code

Tool used

Manual Review

Recommendation

`canRemoveLiquidity()` can be changed to:

```
function canRemoveLiquidity(address target, bytes calldata data)
    internal
    view
    returns (bool, address[] memory, address[] memory)
{
    address[] memory tokensOut = new address[](1);
    tokensOut[0] = target;

    address[] memory tokensIn = new address[](2);
    tokensIn[0] = IStableSwapPool(target).coins(0);
    tokensIn[1] = IStableSwapPool(target).coins(1);
    return (true, tokensIn, tokensOut);
}
```

Issue M-1: Lack of price freshness check in `ChainlinkOracle.solgetPrice()` allows a stale price to be used

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/002-M

Found by

defsec, icedpeachtea, oyc_109, Lambda, 0xNineDec, Avci, ladboy233, JohnSmith, jonatascm, Ruhum, csanuragjain, PwnPatrol, WATCHPUG, 0xNazgul, xiaoming90, 0x52, 0xf15ers, ellahi, pashov, rbserver, GalloDaSballo, Chom, __141345__, cccz, devtooligan, Bahurum, HonorLt, GimelSec, Dravee, Olivierdem

Summary

`ChainlinkOracle` should use the `updatedAt` value from the latestRoundData() function to make sure that the latest answer is recent enough to be used.

Vulnerability Detail

In the current implementation of `ChainlinkOracle.solgetPrice()`, there is no freshness check. This could lead to stale prices being used.

If the market price of the token drops very quickly ("flash crashes"), and Chainlink's feed does not get updated in time, the smart contract will continue to believe the token is worth more than the market value.

Chainlink also advise developers to check for the `updatedAt` before using the price:

> Your application should track the latestTimestamp variable or use the updatedAt value from the latestRoundData() function to make sure that the latest answer is recent enough for your application to use it. If your application detects that the reported answer is not updated within the heartbeat or within time limits that you determine are acceptable for your application, pause operation or switch to an alternate operation mode while identifying the cause of the delay.

And they have this heartbeat concept:

> Chainlink Price Feeds do not provide streaming data. Rather, the aggregator updates its latestAnswer when the value deviates beyond a specified threshold or when the heartbeat idle time has passed. You can find the heartbeat and deviation values for each data feed at data.chain.link or in the Contract Addresses lists.

The `Heartbeat` on Arbitrum is usually `1h`.

Source: https://docs.chain.link/docs/arbitrum-price-feeds/

Impact

A stale price can cause the malfunction of multiple features across the protocol:

1. `_valueInWei()` is using the price to calculate the value of the loan and collateral; A stale price will make the price calculation inaccurate so that certain accounts may not be liquidated when they should be, or be liquidated when they should not be.
2. `ChainlinkOracle.solgetPrice()` is used to calculate the value of various LPTokens (Aave, Balancer, Compound, Curve, and Uniswap). If the price is not accurate, it will lead to a deviation in the LPToken price and affect the calculation of asset prices.
3. Stale asset prices can lead to bad debts to the protocol as the collateral assets can be overvalued, and the collateral value can not cover the loans.

Code Snippet

https://github.com/sentimentxyz/oracle/blob/59b26a3d8c295208437aad36c470386c9729a4bc/src/chainlink/ChainlinkOracle.sol#L49-L59

```
function getPrice(address token) external view virtual returns (uint) {
    (, int answer,,,) =
        feed[token].latestRoundData();

    if (answer < 0)
        revert Errors.NegativePrice(token, address(feed[token]));

    return (
        (uint(answer)*1e18)/getEthPrice()
    );
}
```

https://github.com/sentimentxyz/oracle/blob/59b26a3d8c295208437aad36c470386c9729a4bc/src/chainlink/ChainlinkOracle.sol#L65-L73

```
function getEthPrice() internal view returns (uint) {
    (, int answer,,,) =
        ethUsdPriceFeed.latestRoundData();

    if (answer < 0)
        revert Errors.NegativePrice(address(0), address(ethUsdPriceFeed));

    return uint(answer);
}
```

Tool used

Manual Review

Recommendation

Consider adding the missing freshness check for stale price:

```
function getPrice(address token) external view virtual returns (uint) {
    (, int answer,,,) =
        feed[token].latestRoundData();
    uint validPeriod = feedValidPeriod[token];

    require(block.timestamp - updatedAt < validPeriod, "freshness check failed.")

    if (answer <= 0)
        revert Errors.NegativePrice(token, address(feed[token]));

    return (
        (uint(answer)*1e18)/getEthPrice()
    );
}
```

The `validPeriod` can be based on the `Heartbeat` of the feed.

Issue M-2: Can register a non-allowed collateral as collateral

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/029-M

Found by

Lambda, Bahurum, kirk-baird, bin2chen, xiaoming90, GalloDaSballo

Summary

Some external interactions send tokens to the account, and the token address is not checked before being registered as a collateral for the account. This allows using an arbitrary token as a collateral as long as there is a corresponding oracle.

Vulnerability Detail

In the following controllers, at the line referenced

UniV2Controller.sol #L189
AaveV2Controller.sol #L91
AaveV3Controller.sol #L79
BalancerController.sol#L137

There is an operation with a return value that can be an arbitrary token. No check is made to verify that the tokens in `tokensIn` are activated in `controllerFacade.isTokenAllowed`. No check is made either in `AccountManager.exec()` when calling `_updateTokensIn()` (AccountManager.sol#L305), so the tokens are added as a collateral as long as the oracle for the token in `OracleFacade` is active.

This can be clarified with an example using `UniV2Controller.sol`:

1. Admin sets the oracle for token XYZ, but does not activate yet the token in `controllerFacade.isTokenAllowed` or in `AccountManager.isCollateralAllowed` waiting to clear some security concerns with the token (vulnerabilities, high volatility, low liquidity, ...)
2. Attacker opens an account
3. Attacker provides liquidity to XYZ / ETH pool on UniswapV2
4. Attacker transfers the LP to the account
5. Attacker removes liquidity from the UniswapV2 pool through the account by calling `AccountManager.exec`. Note that there is no check to control whether tokens exiting the account are allowed (UniV2Controller.sol#L175-L190). No checks are performed on tokens sent from the pool to the account, so account

receives WETH + XYZ and also XYZ is registered as an account collateral (AccountManager.sol#L305). An oracle was set for XYZ so the call to `riskEngine.isAccountHealthy(account)` (AccountManager.sol#L310) does not revert and the transaction succeeds.

6. Attacker can take advantage of the aforementioned security concerns to manipulate the price of XYZ, inflate his collateral balance and drain funds from the protocol.

The same behaviour can be reproduced with the other integrations listed above by transfering LP tokens of that particular protocol directly to an account and then removing liquidity through `AccountManager.exec()` to get the pair tokens registered into the account as collateral.

Impact

See point 6. of section above.

Code Snippet

UniV2Controller.sol #L189
AaveV2Controller.sol #L91
AaveV3Controller.sol #L79
BalancerController.sol#L137
AccountManager.sol#L347

Tool used

Manual Review

Recommendation

Check if tokens are allowed as collateral in `_updateTokensIn` (AccountManager.sol#L347-L355)

```
    function _updateTokensIn(address account, address[] memory tokensIn)
        internal
    {
        uint tokensInLen = tokensIn.length;
        for(uint i; i < tokensInLen; ++i) {
+           if (!isCollateralAllowed[tokensIn[i]])
+               revert Errors.CollateralTypeRestricted();
            if (IAccount(account).hasAsset(tokensIn[i]) == false)
                IAccount(account).addAsset(tokensIn[i]);
        }
    }
```

This way no asset can be added as collateral without being allowed.

Issue M-3: AccountManager: Liquidations not possible when transfer fails

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/033-M

Found by

panprog, csanuragjain, Czar102, carrot, PwnPatrol, Lambda, kirk-baird, berndart-mueller, rbserver, Chom, __141345__

Summary

When the transfer of one asset fails, liquidations become impossible.

Vulnerability Detail

`_liquidate` calls `sweepTo`, which iterates over all assets. When one of those transfers fails, the whole liquidation process therefore fails. There are multiple reasons why a transfer could fail: 1.) Blocked addresses (e.g., USDC) 2.) The balance of the asset is 0, but it is still listed under asset. This can be for instance triggered by performing a 0 value Uniswap swap, in which case it is still added to `tokensIn`. Another way to trigger is to call `deposit` with `amt=0` (this is another issue that should be fixed IMO, in practice the assets of an account should not contain any tokens with zero balance) Some tokens revert for zero value transfers (see https://github.com/d-xo/weird-erc20) 3.) Paused tokens 4.) Upgradeable tokens that changed the implementation.

Impact

See above, an account cannot be liquidated. In certain conditions, this might even be triggerable by the user. For instance, a user could try to get on the USDC blacklist to avoid liquidations.

Code Snippet

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/core/AccountManager.sol#L384 https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/core/Account.sol#L166

Tool used

Manual Review

Recommendation

Catch reversions for the transfer and skip this asset (but it could be kept in the assets list to allow retries later on).

Issue M-4: Accounts with ETH loans can not be liquidated if LEther's underlying is set to `address(0)`

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/034-M

Found by

Lambda, WATCHPUG, hansfriese, rbserver, HonorLt, 0xc0ffEE

Summary

Setting `address(0)` as LEther's `underlying` is allowed, and the logic in `AccountManagersettle()` and `RiskEngine_valueInWei()` handles `address(0)` specially, which implies that `address(0)` can be an asset.

However, if LEther's underlying is set to `address(0)`, the accounts with ETH loans will become unable to be liquidated.

Vulnerability Detail

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/AccountManager.sol#L318-L326

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/RiskEngine.sol#L178-L188

Given that at `AccountManager.solL100` in `settle()` and `RiskEngine.solL186` in `_valueInWei()`, they both handled the case that the `asset==address(0)`, and in `Registry.solsetLToken()`, `underlying==address(0)` is allowed:

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/Registry.sol#L95-L105

We assume that `address(0)` can be set as the `underlying` of `LEther`.

In that case, when the user borrows native tokens, `address(0)` will be added to the user's assets and borrows list.

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/AccountManager.sol#L203-L217

```
function borrow(address account, address token, uint amt)
    external
    whenNotPaused
    onlyOwner(account)
{
    if (registry.LTokenFor(token) == address(0))
```

```
        revert Errors.LTokenUnavailable();
    if (!riskEngine.isBorrowAllowed(account, token, amt))
        revert Errors.RiskThresholdBreached();
    if (IAccount(account).hasAsset(token) == false)
        IAccount(account).addAsset(token);
    if (ILToken(registry.LTokenFor(token)).lendTo(account, amt))
        IAccount(account).addBorrow(token);
    emit Borrow(account, msg.sender, token, amt);
}
```

This will later prevent the user from being liquidated because in `riskEngine.isAccountHealthy()`, it calls `_getBalance()` in the for loop of all the assets, which assumes all the assets complies with `IERC20`. Thus, the transaction will revert at L157 when calling `IERC20(address(0)).balanceOf(account)`.

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/AccountManager.sol#L250-L255

```
function liquidate(address account) external {
    if (riskEngine.isAccountHealthy(account))
        revert Errors.AccountNotLiquidatable();
    _liquidate(account);
    emit AccountLiquidated(account, registry.ownerFor(account));
}
```

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/RiskEngine.sol#L150-L161

```
function _getBalance(address account) internal view returns (uint) {
    address[] memory assets = IAccount(account).getAssets();
    uint assetsLen = assets.length;
    uint totalBalance;
    for(uint i; i < assetsLen; ++i) {
        totalBalance += _valueInWei(
            assets[i],
            IERC20(assets[i]).balanceOf(account)
        );
    }
    return totalBalance + account.balance;
}
```

Impact

We noticed that in the deployment documentation, LEther is set to init with WETH as the `underlying`. Therefore, this should not be an issue if the system is being deployed correctly.

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/protocol/deployments/ArbiDeploymentFlow.md#L47-L53

But considering that setting `address(0)` as LEther's `underlying` is still plausible and the potential damage to the whole protocol is high (all the accounts with ETH loans can not be liquidated), we believe that this should be a medium severity issue.

Code Snippet

Tool used

Manual Review

Recommendation

1. Consider removing the misleading logic in `AccountManagersettle()` and `RiskEngine_valueInWei()` that handles `address(0)` as an asset;
2. Consider disallowing adding `address(0)` as `underlying` in `setLToken()`.

Issue M-5: Balances of rebasing tokens aren't properly tracked

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/035-M

Found by

Lambda, JohnSmith, PwnPatrol, lllllll, xiaoming90, ellahi, bytehat

Summary

Rebasing tokens are tokens where `balanceOf()` returns larger amounts over time, due to the addition of interest to each account, or due to airdrops

Vulnerability Detail

Sentiment doesn't properly track balance changes while rebasing tokens are in the borrower's account

Impact

The lender will miss out on gains that should have accrued to them while the asset was lent out. While market-based price corrections may be able to handle interest that is accrued to everyone, market approaches won't work when only subsets of token addresses are given rewards, e.g. an airdrop based on a snapshot of activity that happened prior to the token being lent.

Code Snippet

Lending tracks shares of the LToken: https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/tokens/LToken.sol#L140-L143

But repayment assumes that shares are equal to the same amount, regardless of which address held them, which is not true for airdrops: https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/tokens/LToken.sol#L160-L163

Rebasing tokens are supported, since Aave is a rebasing token: https://github.com/sherlock-audit/2022-08-sentiment/blob/main/controller/src/aave/AaveEthController.sol#L28

Tool used

Manual Review

Recommendation

Adjust share amounts when the account balance doesn't match the share conversion calculation when taking into account gains made by the borrower

Issue M-6: If oracle is set for ERC777 token, re-entrancy is possible to steal all LToken funds

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/076-M

Found by

panprog, xiaoming90, Tutturu, devtooligan, Bahurum, Czar102

Summary

If oracle is set for any `ERC777` or similar token (tokens which call receiver's hook after receiving it), re-entrancy in `Account.sweepTo` allows to borrow funds, which are immediately withdrawn along with all account assets without any health checks, leaving account with 0 assets and big debt, making it possible to drain all LToken funds.

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/core/Account.sol#L163-L174

Vulnerability Detail

If oracle is set for `ERC777` token, it's possible to add these tokens to assets (even if collateral or controller allowed token for them is not set, using the bug in uniswap v2 controller for `removeLiquidity`, which doesn't check if tokenIn is allowed).

Once `ERC777` token is added to assets, malicious user can close his account, which calls `Account.sweepTo`, which has multiple re-entrancy scenarios. As `ERC777` token will call user's hook after receiving these tokens, the following actions are possible in the hook to steal funds:

- User can borrow funds, exit reentrancy, and all borrowed tokens along with all account assets will be immediately transfered to user without any health checks.
- For tokens which were in `assets` list before the `ERC777` token, `hasAsset[]` is set to false even though the token is still in the assets list. Depositing these tokens again will add them to assets again, counting them twice in account balance calculations, which allows to borrow even more funds.
- Ether is transferred back to user after all the `ERC20` tokens, so ether can also be used to borrow funds against, and it will also be transerred back to user along with all the `ERC20` tokens.

List of bugs used in the attack:

1. Uniswap v2 controller in `removeLiquidity` doesn't check if tokens received are allowed. It seems to assume that account can only have allowed uni v2 lp tokens (as it's checked in `addLiquidity`), however any lp tokens can easily be transferred to account directly (not via `exec`). This makes it possible to call `exec` to `removeLiquidity` and add ANY tokens to account's assets list (and if oracle is set for the token, `accountManager.exec` will succeed as it doesn't check for asset tokens to be allowed as collateral)

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/controller/src/uniswap/UniV2Controller.sol#L182-L189

2. `Account.sweepTo` is very vulnerable to re-entrancy. While it assumes that asset tokens are ERC20, if ERC20-compatible tokens with callback hooks are allowed (such as `ERC777` tokens), these tokens can re-enter to steal funds.

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/core/Account.sol#L163-L174

Steps to steal funds from LToken:

1. OpenAccount
2. If `ERC777` token is allowed as collateral, deposit `ERC777` token (such as `imBTC`, which is pegged 1:1 to BTC and is ERC777). If `ERC777` is not allowed as collateral, but the oracle for this token is set up, then use bug 1: 2.1. Deploy uniswap v2 pool `USDC-imBTC`, add liquidity with minimum `USDC` and `imBTC` 2.2. Send uniswap v2 LP token to account's address 2.3. Call `AccountManager.exec` to `removeLiquidity` from uniswap v2 LP token, abusing bug 1 and receiving small amounts of `USDC+imBTC` (`tokensIn` are set and both `USDC` and `imBTC` are added to account's `assets`)
3. Deposit `0DAI` to add `DAI` to account `assets` (this is to include `DAI` in `sweepTo` at the last index, which will make it possible to borrow `DAI` in re-entrancy and immediately receive all of it without health checks)
4. Close account
5. Close account will performs `account.sweepTo` as the last step, where `USDC` is transferred to user, then `imBTC` is transferred to user, which calls `tokensReceived` on user's contract allowing re-entrancy. In `tokensReceived`: 5.1. Call `AccountManager.openAccount` to re-gain access to account (since it's closed) - this restores account access 5.2. Deposit `1ether` to account 5.3. Borrow `4000DAI` 5.4. Exit re-entrancy
6. `sweepTo` continues and sends `4000DAI` to user, then deletes `assets` and sends `1 ether` to user.
7. At this point user has received back its `1ether` and `4000DAI`, leaving account with `0` assets and `4000DAI` debt.

This scenario can be made even more capital efficient to steal more money if user deposits `1000USDC` instead of `1ether`, which will add `USDC` to assets, making his balance `2000` instead of `1000`, allowing to borrow `8000DAI` for the same deposited amount. If this is repeated a few times, it's possible to steal millions with only a small initial capital.

Impact

If any `ERC777` (or similar) token oracle is added, it's very easy to steal all funds from all LTokens which you can borrow from.

Code Snippet

Create folder `./protocol/src/test/integrations/attacks` and put these 2 test files there:

https://gist.github.com/panprog/2f16348325303869f7d84653cf99fba1

Tool used

Manual Review

Recommendation

1. Add ReEntrancy guard from openzeppelin to main entry functions, mostly all functions in `AccountManager`.
2. Add correct token checks to uniswap v2 controller.

Issue M-7: Uniswap contract added to controller doesn't match with function signatures

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/092-M

Found by

PwnPatrol

Summary

In the `ArbiDeploymentFlow.md` doc, it specifies the plan to add the UniV3Controller to controllerFacade, and then update it so it applies to Router (address: `0xE592427A0AEce92De3Edee1F18E0157C05861564`).

However, some of the function signatures of the Router at this address don't match the signatures specified in the controller, so function calls will revert.

Vulnerability Detail

In `UniV3Controller.sol`, the function signatures that are allowed to be used when calling the contract are specified.

However, these signatures differ between the two Uniswap V3 router contracts due to a change in the parameters for the arguments (in all four functions, the original router included a uint256 for duration, and the Router 2 contract does not).

As a result, the function signatures in the controller do not line up with the signatures in the Router contract that is specified in the deployment doc:

- exactInputSingle should be 0x414bf389, not 0x04e45aaf
- exactOutputSingle should be 0xdb3e2198, not 0x5023b4df
- exactInput should be 0xc04b8d59, not 0xb858183f
- exactOutput should be 0xf28c0498, not 0x09b81346

Impact

Users attempting to interact with Uniswap V3 through the UniV3Controller will have their calls rejected due to mismatched function signatures.

Code Snippet

```
/// @notice
↪  exactInputSingle((address,address,uint24,address,uint256,uint256,uint160))
↪  function signature
bytes4 constant EXACT_INPUT_SINGLE = 0x04e45aaf;

/// @notice
↪  exactOutputSingle((address,address,uint24,address,uint256,uint256,uint160))
↪  function signature
bytes4 constant EXACT_OUTPUT_SINGLE = 0x5023b4df;

/// @notice exactInput((bytes,address,uint256,uint256)) function signature
bytes4 constant EXACT_INPUT = 0xb858183f;

/// @notice exactOutput((bytes,address,uint256,uint256)) function signature
bytes4 constant EXACT_OUTPUT = 0x09b81346;
```

Tool Used

Manual Review

Recommendation

Use Router 2 (address: 0x68b3465833fb72A70ecDF485E0e4C7bD8665Fc45) when calling `controllerFacade.updateController()` to connect UniV3Controller to the correct router.

Alternatively, add the extra function signatures in `UniV3Controller.sol` so the controller is able to work on either Router.

Issue M-8: Missing revert keyword

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/097-M

Found by

PwnPatrol

Summary

Missing `revert` keyword in `functionDelegateCall` bypasses an intended safety check, allowing the function to fail silently.

Vulnerability Detail

In the helper function `functionDelegateCall`, there is a check to confirm that the target being called is a contract.

```
if (!isContract(target)) Errors.AddressNotContract;
```

However, there is a typo in the check that is missing the `revert` keyword.

As a result, non-contracts can be submitted as targets, which will cause the delegatecall below to return success (because EVM treats no code as STOP opcode), even though it doesn't do anything.

```
(bool success, ) = target.delegatecall(data);
require(success, "CALL_FAILED");
```

Impact

The code doesn't accomplish its intended goal of checking to confirm that only contracts are passed as targets, so delegatecalls can silently fail.

Code Snippet

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/utils/Helpers.sol#L66-L73

```
function functionDelegateCall(
    address target,
    bytes calldata data
) internal {
    if (!isContract(target)) Errors.AddressNotContract;
```

```
    (bool success, ) = target.delegatecall(data);
    require(success, "CALL_FAILED");
}
```

Tool used

Manual Review

Recommendation

Add missing `revert` keyword to L70 of Helpers.sol.

```
if (!isContract(target)) revert Errors.AddressNotContract;
```

Issue M-9: Protocol Reserve Within A LToken Vault Can Be Lent Out

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/122-M

Found by

xiaoming90

Summary

Protocol reserve, which serves as a liquidity backstop or to compensate the protocol, within a LToken vault can be lent out to the borrowers.

Vulnerability Detail

The purpose of the protocol reserve within a LToken vault is to compensate the protocol or serve as a liquidity backstop. However, based on the current setup, it is possible for the protocol reserve within a Ltoken vault to be lent out.

The following functions within the `LToken` contract show that the protocol reserve is intentionally preserved by removing the protocol reserve from the calculation of total assets within a LToken vault. As such, whenever the Liquidity Providers (LPs) attempt to redeem their LP token, the protocol reserves will stay intact and will not be withdrawn by the LPs.

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/tokens/LToken.sol#L191

```
function totalAssets() public view override returns (uint) {
    return asset.balanceOf(address(this)) + getBorrows() - getReserves();
}
```

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/tokens/LToken.sol#L195

```
function getBorrows() public view returns (uint) {
    return borrows + borrows.mulWadUp(getRateFactor());
}
```

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/tokens/LToken.sol#L176

```
function getReserves() public view returns (uint) {
    return reserves + borrows.mulWadUp(getRateFactor())
```

```
        .mulWadUp(reserveFactor);
}
```

However, this measure is not applied consistently across the protocol. The following `lendTo` function shows that as long as the borrower has sufficient collateral to ensure their account remains healthy, the borrower could borrow as many assets from the LToken vault as they wish.

In the worst-case scenario, the borrower can borrow all the assets from the LToken vault, including the protocol reserve.

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/tokens/LToken.sol#L128

```
File: LToken.sol
121:     /**
122:         @notice Lends a specified amount of underlying asset to an account
123:         @param account Address of account
124:         @param amt Amount of token to lend
125:         @return isFirstBorrow Returns if the account is borrowing the asset
 ↪  for
126:             the first time
127:     */
128:     function lendTo(address account, uint amt)
129:         external
130:         whenNotPaused
131:         accountManagerOnly
132:         returns (bool isFirstBorrow)
133:     {
134:         updateState();
135:         isFirstBorrow = (borrowsOf[account] == 0);
136:
137:         uint borrowShares;
138:         require((borrowShares = convertAssetToBorrowShares(amt)) != 0,
 ↪  "ZERO_BORROW_SHARES");
139:         totalBorrowShares += borrowShares;
140:         borrowsOf[account] += borrowShares;
141:
142:         borrows += amt;
143:         asset.safeTransfer(account, amt);
144:         return isFirstBorrow;
145:     }
```

Impact

The purpose of the protocol reserve within a LToken vault is to compensate the protocol or serve as a liquidity backstop. Without the protocol reserve, the protocol will

become illiquidity, and there is no fund to compensate the protocol.

Recommendation

Consider updating the `lendTo` function to ensure that the protocol reserve is preserved and cannot be lent out. If the underlying asset of a LToken vault is less than or equal to the protocol reserve, the lending should be paused as it is more important to preserve the protocol reserve compared to lending them out.

```
function lendTo(address account, uint amt)
    external
    whenNotPaused
    accountManagerOnly
    returns (bool isFirstBorrow)
{
    updateState();
    isFirstBorrow = (borrowsOf[account] == 0);

    require

    uint borrowShares;
    require((borrowShares = convertAssetToBorrowShares(amt)) != 0,
↪   "ZERO_BORROW_SHARES");
    totalBorrowShares += borrowShares;
    borrowsOf[account] += borrowShares;

    borrows += amt;
    asset.safeTransfer(account, amt);

+   require(asset.balanceOf(address(this)) >= getReserves(), "Not enough
↪   liquidity for lending")

    return isFirstBorrow;
}
```

Issue M-10: ERC4626Oracle Vulnerable To Price Manipulation

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/133-M

Found by

xiaoming90, lllllll

Summary

ERC4626 oracle is vulnerable to price manipulation. This allows an attacker to increase or decrease the price to carry out various attacks against the protocol.

Vulnerability Detail

The `getPrice` function within the `ERC4626Oracle` contract is vulnerable to price manipulation because the price can be increased or decreased within a single transaction/block.

Based on the `getPrice` function, the price of the LP token of an ERC4626 vault is dependent on the `ERC4626.previewRedeem` and `oracleFacade.getPrice` functions. If the value returns by either `ERC4626.previewRedeem` or `oracleFacade.getPrice` can be manipulated within a single transaction/block, the price of the LP token of an ERC4626 vault is considered to be vulnerable to price manipulation.

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/oracle/src/erc4626/ERC4626Oracle.sol#L8

```
File: ERC4626Oracle.sol
35:     function getPrice(address token) external view returns (uint) {
36:         uint decimals = IERC4626(token).decimals();
37:         return IERC4626(token).previewRedeem(
38:             10 ** decimals
39:         ).mulDivDown(
40:             oracleFacade.getPrice(IERC4626(token).asset()),
41:             10 ** decimals
42:         );
43:     }
```

It was observed that the `ERC4626.previewRedeem` couldbe manipulated within a single transaction/block. As shown below, the `previewRedeem` function will call the `convertToAssets` function. Within the `convertToAssets`, the number of assets per share is calculated based on the current/spot total assets and current/spot supply that can be increased or decreased within a single block/transaction by calling the vault's

deposit, mint, withdraw or redeem functions. This allows the attacker to artificially inflate or deflate the price within a single block/transaction.

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/tokens/utils/ERC4626.sol#L154

```
File: ERC4626.sol
154:     function previewRedeem(uint256 shares) public view virtual returns
 ↪  (uint256) {
155:         return convertToAssets(shares);
156:     }
```

https://github.com/sherlock-audit/2022-08-sentiment/blob/main/protocol/src/tokens/utils/ERC4626.sol#L132

```
File: ERC4626.sol
132:     function convertToAssets(uint256 shares) public view virtual returns
 ↪  (uint256) {
133:         uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply
 ↪  is non-zero.
134:
135:         return supply == 0 ? shares : shares.mulDivDown(totalAssets(),
 ↪  supply);
136:     }
```

Impact

The attacker could perform price manipulation to make the apparent value of an asset to be much higher or much lower than the true value of the asset. Following are some risks of price manipulation:

- An attacker can increase the value of their collaterals to increase their borrowing power so that they can borrow more assets than they are allowed from Sentiment.
- An attacker can decrease the value of some collaterals and attempt to liquidate another user account prematurely.

Recommendation

Avoid using `previewRedeem` function to calculate the price of the LP token of an ERC4626 vault. Consider implementing TWAP so that the price cannot be inflated or deflated within a single block/transaction or within a short period of time.

Issue M-11: Delisted assets can still be deposited and borrowed against by accounts that already have them

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/154-M

Found by

0x52, Kumpa, devtooligan, WATCHPUG

Summary

Delisting an asset does not prevent accounts, that already contain the asset from depositing more. It blocks the deposit function but users can sidestep but sending tokens directly to their account

Vulnerability Detail

AccountManger.sol#deposit attempts to block deposits from assets that are not on the list of supported collateral. When calculating the health of the account, the total balance of the account address is considered. An account that already has the asset on it's assets list doesn't need to use AccountManger.sol#deposit because they can transfer the tokens directly to the account contract. This means that these account can continue to add to and use delisted assets as collateral.

Impact

One of two scenarios depending on actions taken by the protocol. If the asset is delisted from accountManager.sol and the oracle is removed then all users with loans taken against the asset will likely be immediately liquidated, which is highly unfair to users. If the asset is just delisted from accountManager.sol then existing users that already have the asset would be able to continue using the asset to take loans. If the reason an asset is being delisted is to prevent a vulnerability then the exploit would still be able to happen due to these accounts sidestepping deposit restrictions.

Code Snippet

AccountManager.sol#L162-L172

Tool used

Manual Review

Recommendation

Calculations for account health should be split into two distinct categories. When calculating the health of a position post-action, unsupported assets should not be considered in the total value of the account. When calculating the health of a position for liquidation, all assets on the asset list should be considered. This prevent any new borrowing against a delisted asset but doesn't risk all affected users being liquidated unfairly.

Issue M-12: M-03 YTokenOracle doesn't account for losses when pricing the yToken

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/204-M

Found by

GalloDaSballo

Summary

Yearn vaults use multiple strategies, some of them can incurr a loss, the loss will be charged to the caller when withdrawing (the caller being the protocol).

The Ytoken.getPrice incorrectly assumes that a withdrawal of 10 ** decimals is equivalent to a bigger withdrawal.

Because of the Yearn V2 codebase, we can see that a loss may happen on withdraw: https://github.com/yearn/yearn-vaults/blob/efb47d8a84fcb13ceebd3ceb11b126b323bcc05d/contracts/Vault.vy#L1074

Meaning that using the 10 ** decimals price for quoting is incorrect as it doesn't represent the total liquidatable assets, which may be very different (especially for tokens that use multiple strategies, especially levered strategies such as Levered AAVE, Levered COMP etc..)

Vulnerability Detail

- Attacker knows vault can only withdraw up to 50% of the underlying before getting rekt
- Attacker borrows 100% of token value, the oracle allows this as the oracle only checks for the price of 10 ** decimals tokens
- Attacker purposefully get's liquidated
- The system withdraws from the yearn vault and incurs a loss
- Attacker has ran away with more value than intended
- Additionally, self-liquidation may be used as if the Yearn Vault is using any liquidity pool to liquidate to token, the imbalance caused by the withdrawal could itself create an opportunity for an arbitrage

Impact

Oracle will allow to borrow more than what is liquidatable from the yVault

Tool used

Manual Review

Recommendation

Refactor `getPrice(address)` to `getPrice(address,amount)` and pass in the exact amount to ensure it can be liquidated fully without a loss

Alternatively cap the Yearn tokens to a massively small LTV (35% is probably as high as you should go without simming on a token by token basis)

Issue M-13: M-05 Yearn `withdraw(uint)` selector may backfire

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/208-M

Found by

GalloDaSballo

Summary

The YearnController is using:

```
/// @notice withdraw(uint256) function signature
 bytes4 constant WITHDRAW = 0x2e1a7d4d;
```

Which is the selector for withdraw(amount, msg.sender, 1) where the max_loss is 1bps

See Yearn Code: https://github.com/yearn/yearn-vaults/blob/efb47d8a84fcb13ceebd3ceb11b126b323bcc05d/contracts/Vault.vy#L1010

Vulnerability Detail

On one hand the selector will allow for loss, if you'd want to have no loss you should change it to a function call with a hardcoded 0 value for `maxLoss`

On the other hand, if the account has too big of a position in a Yearn Vault (e.g. more than 10% of the vault), then a withdrawal may simply not be possible as the function will always revert.

You can monitor the limits on `https://yearn.watch/`, per the Yearn withdraw code, the function will go through the withdrawal queue, adding up losses (if any) to the caller.

Because the hardcoded selector will not offer any flexibility to the `maxLoss` parameter, in any scenario in which the losses are above 1 bps, the function will simply revert, preventing any withdrawal.

Tool used

Manual Review

Recommendation

Consider adding support for a more complete withdraw that allows to change the `maxLoss` parameter

Issue M-14: `Reserves` should not be considered part of the available liquidity while calculating the interest rate

Source: https://github.com/sherlock-audit/2022-08-sentiment-judging/tree/main/266-M

Found by

WATCHPUG

Summary

The implementation is different from the documentation regarding the interest rate formula.

Vulnerability Detail

The formula given in the docs:

> Calculates Borrow rate per second:
>
> $$BorrowRatePerSecond = c3 \cdot (util \cdot c1 + util^{32} \cdot c1 + util^{64} \cdot c2) \div secsPerYear$$
>
> where, $util = borrows \div (liquidity - reserves + borrows)$

$$util = borrows \div (liquidityreserves + borrows)$$

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/tokens/LToken.sol#L217-L227

```
function getRateFactor() internal view returns (uint) {
    return (block.timestamp == lastUpdated) ?
        0 :
        ((block.timestamp - lastUpdated)*1e18)
        .mulWadUp(
            rateModel.getBorrowRatePerSecond(
                asset.balanceOf(address(this)),
                borrows
            )
        );
}
```

However, the current implementation is taking all the balance as the liquidity:

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/DefaultRateModel.sol#L51-L68

```
function getBorrowRatePerSecond(
    uint liquidity,
    uint borrows
)
    external
    view
    returns (uint)
{
    uint util = _utilization(liquidity, borrows);
    return c3.mulDivDown(
        (
            util.mulWadDown(c1)
            + util.rpow(32, SCALE).mulWadDown(c1)
            + util.rpow(64, SCALE).mulWadDown(c2)
        ),
        secsPerYear
    );
}
```

https://github.com/sentimentxyz/protocol/blob/4e45871e4540df0f189f6c89deb8d34f24930120/src/core/DefaultRateModel.sol#L70-L77

```
function _utilization(uint liquidity, uint borrows)
    internal
    pure
    returns (uint)
{
    uint totalAssets = liquidity + borrows;
    return (totalAssets == 0) ? 0 : borrows.divWadDown(totalAssets);
}
```

Impact

Per the docs, when calculating the interest rate, `util` is the ratio of available liquidity to the `borrows`, available liquidity should not include reserves.

The current implementation is using all the balance as the `liquidity`, this will make the interest rate lower than expectation.

PoC

Given:

- `asset.address(this)+borrows=10000`

- `reserves=1500,borrows=7000`

Expected result:

When calculating `getRateFactor()`, available liquidity should be `asset.balanceOf(address(this))-reserves=1500,util=7000/8500=0.82`, `getBorrowRatePerSecond()=9114134329`

Actual result:

When calculating `getRateFactor()`, `asset.balanceOf(address(this))=3000,util=0.7e18`, `getBorrowRatePerSecond()=7763863430`

The actual interest rate is only