SHERLOCK

Security Review For AXION

axion

Public Contest Prepared For: AXION
Lead Security Expert: pkqs90
Date Audited: October 10 - October 15, 2024

Introduction

A Stablecoin built for DEFI, built on DEFI. Collateral is deployed in liquidity pools onchain: the AMO maintains peg while being profitable.

Scope

Repository: AXION-MONEY/liquidity-amo

Branch: main

Audited Commit: 9a9adab905878a3a8c4fbe7c0851354185d8466a

Final Commit: ac8073c1199a49e30ff020b1005899d707ff1c48

Repository: AXION-MONEY/solidly-utils

Branch: main

Audited Commit: 7946d226cc2c14159a6a2bda01ede157e2199f21

Final Commit: a80ff65f001c6ff0006b37443a4ae0f09acb62e8

For the detailed scope, see the contest details.

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.

Issues Found

High	Medium
4	6

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

pkqs90	Kirkeelee	Naresh
0x37	yuza101	RadCet
spark1	hunter_w3b	wellbyt3
s1ce	ABA	ZanyBonzy
carrotsmuggler	FonDevs	Atharv
vinica_boy	UsmanAtique	isagixyz
KupiaSec	calc1f4r	AuditorPraise
y4y	Bigsam	HackTrace
Greese	durov	dany.armstrong90
PNS	unnamed	0xnbvc

Issue H-1: Boost buyback burns incorrect amount of liquidity

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/114

Found by

0x37, carrotsmuggler, pkqs90, s1ce, spark1, vinica_boy

Summary

The function `_unfarmBuyBurn` in the V3AMO contract is a public function open to everyone and calculates the amount of liquidity to burn from the pool. This function basically burns LP positions to take out liquidity and uses the usd to buy up boost tokens and burns them to raise the price of boost tokens.

The issue is in the `_unfarmBuyBurn` function when it tries to estimate how much liquidity needs to be taken out.

https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/V3AMO.sol#L320-L326

As seen above, first the token reserves of the pool are checked. Then, the `liquidity` to be burnt is calculated from the difference of the reserves.

```
liquidity = (totalLiquidity * (boostBalance - usdBalance)) / (boostBalance +
↪  usdBalance);
liquidity = (liquidity * LIQUIDITY_COEFF) / FACTOR;
```

However, this calculation is not valid for V3/CL pools. This is because in V3 pools, single sided liquidity is allowed which adds to the `totalLiquidity` count, but increases the reserves of only 1 token. If a user adds liquidity at a tick lower than the current price, they will be adding only usd to the pool.

For example, lets say the price currently is below peg, at 0.9. Say there are 1000 boost and 900 usd tokens in the pool, similar to a V2 composition. Now, since its a V3 pool, a user can come in and add 100 usd to the pool at a price of 0.5. Since this price is lower than the spot price, only usd will be needed to open this position. Now, the total reserves of both boost and usd are 1000 each, so the calculated `liquidity` amount to be removed will be 0.

Thus the `liquidity` calculated in the contract has absolutely no meaning since it uses the reserves to calculate it, which is not valid for V3 pools. In the best case scenario, this will cause the function to revert and not work. In the worst case scenario, the liquidity calculated will be overestimated and the price will be pushed up even above the peg price. This is possible if users add single sided `boost` to the pool, increasing the `liquidity` amount calculated without changing the price. In this case, the contract assets will be used to force the boost token above peg, and malicious actors can buy the boost token before and sell it after for a handy profit.

Root Cause

The main cause is that `liquidity` is calculated from the reserves. This is not valid for V3, since it can have single sided liquidity, and thus the reserves does not serve as an indicator of price or in this case the deviation from the peg.

Internal pre-conditions

None

External pre-conditions

Any user can add boost-only liquidity to make the contract overestimate the amount of liquidity it needs to burn

Attack Path

Users can add boost-only liquidity to make the contract overestimate the amount of liquidity it needs to burn. When extra liquidity is burnt and extra boost is bought back and burnt, the price will be pushed up even above the peg price. Users can buy before trigerring this and sell after for profit.

Impact

Price can be pushed above the peg price

PoC

None

Mitigation

Use the `quoteSwap` function to calculate how much needs to be swapped until the target price is hit.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/7

Issue H-2: V2AMO is not compatible with Velodrome/Aerodrome.

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/239

Found by

0x37, Greese, KupiaSec, PNS, carrotsmuggler, pkqs90, y4y

Summary

According to the docs, the Dex scope for V2 includes Velodrome/Aerodrome.

> We expect the V2 tech-implementation work with the “classic” pools on the following Dexes: Velodrome, Aerodrome, Thena, Equalizer (Fantom/Sonic/Base), Ramses and forks (legacy pools), Tokan

However, for Velodrome/Aerodrome implementations, the current `V2AMO` is not compatible.

Root Cause

There are two parts of integration with Velodrome/Aerodrome that are buggy:

1. Gauge
2. Router

Let's go through them one by one (Note, since Velodrome and Aerodrome have basically the same code, I will only post Aerodrome code):

1. Gauge The main difference is in the `getReward()` function.

Aerodrome interface: https://github.com/aerodrome-finance/contracts/blob/main/contracts/interfaces/IGauge.sol

```
interface IGauge {
    ...
    function getReward(address _account) external;
    ...
}
```

SolidiyV2AMO interface: https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/interfaces/v2/IGauge.sol#L4

```
interface IGauge {
    ...
    function getReward(address account, address[] memory tokens) external;

    function getReward(uint256 tokenId) external;

    function getReward() external;
    ...
}
```

2. Router The main difference is:

1. Aerodrome uses `poolFor` instead of `pairFor` when querying a pool/pair.
2. The `Route` struct is implemented differently, and is used when performing swap

Aerodrome interface: https://github.com/aerodrome-finance/contracts/blob/main/contracts/interfaces/IRouter.sol#L6

```
interface IRouter {
    struct Route {
        address from;
        address to;
        bool stable;
        address factory;
    }

    function poolFor(
        address tokenA,
        address tokenB,
        bool stable,
        address _factory
    ) external view returns (address pool);

    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        Route[] calldata routes,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
    ...
}
```

SolidiyV2AMO interface: https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/interfaces/v2/IRouter.sol#L4

```
interface IRouter {
    struct route {
        address from;
        address to;
        bool stable;
    }

    function pairFor(address tokenA, address tokenB, bool stable) external view
 ↪  returns (address pair);

    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        route[] memory routes,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
    ...
}
```

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

N/A

Impact

V2AMO does not work with Aerodrome/Velodrome as expected.

PoC

N/A

Mitigation

N/A

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/9

Issue H-3: V3AMO integration with V3 does not collect LP fees when burning liquidity.

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/242

Found by

pkqs90

Summary

V3AMO supplies liquidity for the Boost-USD pool. When users call `unfarmBuyBurn` to burn liquidity from V3AMO, the LP fees should be collected as well. However, the current integration does not correctly collect fees for V3, and the fees are stuck forever.

Root Cause

Let's see the V3 implementation for `burnAndCollect()` function. https://ftmscan.com/address/0x54a571D91A5F8beD1D56fC09756F1714F0cd8aD9#code (This is taken from notion docs.)

Even though V3 is a fork of UniswapV3, there is a significant difference. In UniswapV3, the liquidity fees are calculated within the Pool, and can be collected via the Pool. However, in V3, the fees are not updated at all (In the following code, where the fees should be updated in Position.sol as done by UniswapV3, you can see the fee update code is fully removed).

Also, according to the V3 docs https://docs..com/v3/rewards-distributor, all fees (and bribes) distribution have been moved into the `RewardsDistributor.sol` contract, and distributed via merkel proof. This means calling `burnAndCollect()` does not allow us to collect the LP fees anymore, and that we need to call separate function for V3AMO in order to retrieve the LP fees.

V3Pool.sol

```
    function burnAndCollect(
        address recipient,
        int24 tickLower,
        int24 tickUpper,
        uint128 amountToBurn,
        uint128 amount0ToCollect,
        uint128 amount1ToCollect
    )
```

```
        external
        override
        returns (uint256 amount0FromBurn, uint256 amount1FromBurn, uint128
↪   amount0Collected, uint128 amount1Collected)
    {
        (amount0FromBurn, amount1FromBurn) = _burn(tickLower, tickUpper,
↪   amountToBurn);
        (amount0Collected, amount1Collected) = _collect(
            recipient,
            tickLower,
            tickUpper,
            amount0ToCollect,
            amount1ToCollect
        );
    }
    function _burn(
        int24 tickLower,
        int24 tickUpper,
        uint128 amount
    ) private lock returns (uint256 amount0, uint256 amount1) {
@>      (Position.Info storage position, int256 amount0Int, int256 amount1Int) =
↪   _modifyPosition(
            ModifyPositionParams({
                owner: msg.sender,
                tickLower: tickLower,
                tickUpper: tickUpper,
                liquidityDelta: -int256(amount).toInt128()
            })
        );

        amount0 = uint256(-amount0Int);
        amount1 = uint256(-amount1Int);

        if (amount0 > 0 || amount1 > 0) {
@>          (position.tokensOwed0, position.tokensOwed1) = (
                position.tokensOwed0 + uint128(amount0),
                position.tokensOwed1 + uint128(amount1)
            );
        }
        emit Burn(msg.sender, tickLower, tickUpper, amount, amount0, amount1);
    }
    function _modifyPosition(
        ModifyPositionParams memory params
    ) private returns (Position.Info storage position, int256 amount0, int256
↪   amount1) {
        checkTicks(params.tickLower, params.tickUpper);

        Slot0 memory _slot0 = slot0; // SLOAD for gas optimization
@>      position = _updatePosition(params.owner, params.tickLower,
 ↪  params.tickUpper, params.liquidityDelta);
        ...
    }
    function _updatePosition(
        address owner,
        int24 tickLower,
        int24 tickUpper,
        int128 liquidityDelta
    ) private returns (Position.Info storage position) {
        ...
@>      position.update(liquidityDelta);
        ..
    }
```

V3 Position.sol

```
/// @notice Updates the liquidity amount associated with a user's position
/// @param self The individual position to update
/// @param liquidityDelta The change in pool liquidity as a result of the position
 ↪  update
function update(Info storage self, int128 liquidityDelta) internal {
    // @audit-note: Fees should be accumulated in UniswapV3. But in V3, this is
 ↪  removed.
    if (liquidityDelta != 0) {
        self.liquidity = LiquidityMath.addDelta(self.liquidity, liquidityDelta);
    }
}
```

V3AMO implementation

```
    function _unfarmBuyBurn(
        uint256 liquidity,
        uint256 minBoostRemove,
        uint256 minUsdRemove,
        uint256 minBoostAmountOut,
        uint256 deadline
    )
        internal
        override
        returns (uint256 boostRemoved, uint256 usdRemoved, uint256 usdAmountIn,
 ↪  uint256 boostAmountOut)
    {
        (uint256 amount0Min, uint256 amount1Min) = sortAmounts(minBoostRemove,
 ↪  minUsdRemove);
        // Remove liquidity and store the amounts of USD and BOOST tokens received
        (
            uint256 amount0FromBurn,
            uint256 amount1FromBurn,
            uint128 amount0Collected,
            uint128 amount1Collected
@>      ) = IV3Pool(pool).burnAndCollect(
                address(this),
                tickLower,
                tickUpper,
                uint128(liquidity),
                amount0Min,
                amount1Min,
                type(uint128).max,
                type(uint128).max,
                deadline
            );
        ...
    }
```

- https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/V3AMO.sol#L235

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

N/A

Impact

LP Fees are not retrievable for V3AMO.

PoC

N/A

Mitigation

Add a function to call the `RewardsDistributor.sol` for V3 to retrieve the LP fees. This can be an independent function, since not all V3 forks may support this feature.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/10

Issue H-4: Liquidity is incorrectly calculated during `addLiquidity()` for V3AMO, causing DoS.

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/280

Found by

0x37, pkqs90, s1ce, sparkl

Summary

V3 has the same liquidity calculation as UniswapV3. Currently, when adding liquidity, the liquidity calculation is wrong, and may lead to DoS in some cases.

Root Cause

When calling `addLiquidity`, the amount of liquidity that is suppose to add is calculated by `liquidity=(usdAmount*currentLiquidity)/IERC20Upgradeable(usd).balanceOf(pool);`. This is incorrect in the terms of UniswapV3, because there may be multiple tickLower/tickUpper positions covering the current tick.

Also, since anyone can add a LP position to the pool, so attackers can easily DoS this function.

Consider an attacker adds an unbalanced LP position that deposits a lot of Boost tokens but doesn't deposit USD tokens. This would increase the total liquidity, and inflate the amount of `liquidity` calculated in the above formula, which would lead to an increase of USD tokens required to mint the liquidity.

When the amount of requried USD token is above the approved `usdAmount`, the liquidity minting would fail.

See the following PoC section for a more detailed example.

```
    function _addLiquidity(
        uint256 usdAmount,
        uint256 minBoostSpend,
        uint256 minUsdSpend,
        uint256 deadline
    ) internal override returns (uint256 boostSpent, uint256 usdSpent, uint256
 ↪  liquidity) {
        // Calculate the amount of BOOST to mint based on the usdAmount and
 ↪  boostMultiplier
        uint256 boostAmount = (toBoostAmount(usdAmount) * boostMultiplier) / FACTOR;
        // Mint the specified amount of BOOST tokens to this contract's address
        IMinter(boostMinter).protocolMint(address(this), boostAmount);

        // Approve the transfer of BOOST and USD tokens to the pool
        IERC20Upgradeable(boost).approve(pool, boostAmount);
@>      IERC20Upgradeable(usd).approve(pool, usdAmount);

        (uint256 amount0Min, uint256 amount1Min) = sortAmounts(minBoostSpend,
 ↪  minUsdSpend);

@>      uint128 currentLiquidity = IV3Pool(pool).liquidity();
@>      liquidity = (usdAmount * currentLiquidity) /
 ↪  IERC20Upgradeable(usd).balanceOf(pool);

        // Add liquidity to the BOOST-USD pool within the specified tick range
        (uint256 amount0, uint256 amount1) = IV3Pool(pool).mint(
            address(this),
            tickLower,
            tickUpper,
            uint128(liquidity),
            amount0Min,
            amount1Min,
            deadline
        );
        ...
    }
```

- https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/V3AMO.sol#L186

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

Attackers can brick addLiquidity function by depositing LP.

Impact

Attackers can deposit LP to make add liquidity fail, which also makes `mintSellFarm()` fail. This is an important feature to keep Boost/USD pegged, thus a high severity issue.

This is basically no cost for attackers since the Boost/USD will always go back to 1:1 so no impermanent loss is incurred.

PoC

Add the following code in V3AMO.test.ts. It does the following:

1. Add unbalanced liquidity so that total liquidity increases, but USD.balanceOf(pool) does not increase.
2. Mint some USD to V3AMO for adding liquidity.
3. Try to add liquidity, but it fails due to incorrect liquidity calculation (tries to add too much liquidity for not enough USD tokens).

```
it("Should execute addLiquidity successfully", async function() {
  // Step 1: Add unbalanced liquidity so that total liquidity increases, but
↪   USD.balanceOf(pool) does not increase.
  {
    // -276325 is the current slot0 tick.
    console.log(await pool.slot0());
    await boost.connect(boostMinter).mint(admin.address, boostDesired * 100n);
    await testUSD.connect(boostMinter).mint(admin.address, usdDesired * 100n);
    await boost.approve(poolAddress, boostDesired * 100n);
    await testUSD.approve(poolAddress, usdDesired * 100n);
    console.log(await boost.balanceOf(admin.address));
    console.log(await testUSD.balanceOf(admin.address));
    await pool.mint(
      amoAddress,
      -276325 - 10,
      tickUpper,
      liquidity * 3n,
      0,
      0,
      deadline
    );
    console.log(await boost.balanceOf(admin.address));
    console.log(await testUSD.balanceOf(admin.address));
  }

  // Step 2: Mint some USD to V3AMO for adding liquidity.
  await testUSD.connect(admin).mint(amoAddress, ethers.parseUnits("1000", 6));
  const usdBalance = await testUSD.balanceOf(amoAddress);

  // Step 3: Add liquidity fails due to incorrect liquidity calculation.
  await expect(V3AMO.connect(amo).addLiquidity(
    usdBalance,
    1,
    1,
    deadline
  )).to.emit(V3AMO, "AddLiquidity");
});
```

Mitigation

Use the UniswapV3 library for calculating liquidity: https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/LiquidityAmounts.sol#L56

```
function getLiquidityForAmounts(
    uint160 sqrtRatioX96,
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    uint256 amount0,
    uint256 amount1
) internal pure returns (uint128 liquidity) {
    if (sqrtRatioAX96 > sqrtRatioBX96) (sqrtRatioAX96, sqrtRatioBX96) =
 ↪  (sqrtRatioBX96, sqrtRatioAX96);

    if (sqrtRatioX96 <= sqrtRatioAX96) {
        liquidity = getLiquidityForAmount0(sqrtRatioAX96, sqrtRatioBX96, amount0);
    } else if (sqrtRatioX96 < sqrtRatioBX96) {
        uint128 liquidity0 = getLiquidityForAmount0(sqrtRatioX96, sqrtRatioBX96,
 ↪  amount0);
        uint128 liquidity1 = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioX96,
 ↪  amount1);

        liquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;
    } else {
        liquidity = getLiquidityForAmount1(sqrtRatioAX96, sqrtRatioBX96, amount1);
    }
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/12

Issue M-1: Boost can be sold under peg despite comments and code attempting to prevent it

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/36

Found by

spark1, vinica_boy

Summary

Boost can be sold under peg despite comments and code attempting to prevent it. This is defined in the comments https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/MasterAMO.sol#L148-L150 and readme " There are, however, some hard-coded limitations –– mainly to ensure that **even admins can only** buyback BOOST below peg and **sell it above peg**. These are doctsringed."

Root Cause

The only apparent check to prevent boost from being sold below peg is a simple if statement:

https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/V2AMO.sol#L171

```
if (minUsdAmountOut < toUsdAmount(boostAmount)) minUsdAmountOut =
↪  toUsdAmount(boostAmount);
```

However this does not prevent boost from being sold below peg. Consider pool with 90 boost and 110 usdc (ignore fees for now). 1 boost > 1 usdc. If _mintAndSellBoost is called with 20 boost, we will get end result 110 boost and 90 usdc. 20 boost were sold for 20 usdc. However, some of the boost were sold below the peg. Consider the first boost we sold. It was above the peg. At about 99.498 boost and 99.498 usdc the pool will be balanced. However, once we continue selling boost we will be selling below the peg.

The admin can call mintAndSellBoost directly. https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/MasterAMO.sol#L155-L168

The following affiliated check does nothing as without fee on transfer it will always pass. https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/V2AMO.sol#L186-L188

```
// we check that selling BOOST yields proportionally more USD
if (usdAmountOut != usdBalanceAfter - usdBalanceBefore)
    revert UsdAmountOutMismatch(usdAmountOut, usdBalanceAfter - usdBalanceBefore);
```

This also applies to mintSellFarm as it does the same call with toUsdAmount(boostAmount) https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/V2AMO.sol#L349.

This may not apply to V3 AMO depending on if targetSqrtPriceX96 is set correctly: https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/V3AMO.sol#L141-L160

This contradicts the comments and readme stating that boost cannot be sold below peg by the AMO.

Internal pre-conditions

no preconditions

External pre-conditions

no preconditions

Attack Path

Not an attack

Impact

The protocol loses funds equivalent to the area under the curve of boost sold below peg. In large depeg this could be a huge amount of money. In the 90 - 110 example the total loss is 10.502 - 9.498 = 1.004. The protocol has lost 5% of the funds used in _mintAndSellBoost. High severity as it has large loss and violates important invariant specified explicitly in readme.

PoC

Not required according to the terms

Mitigation

Check that you can only sell to the sqrt K 1:1 balanced price.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/4

Issue M-2: V2AMO and V3AMO: USDT Approval Logic Causes Reversion

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/47

Found by

0x37, Naresh, RadCet, ZanyBonzy, unnamed, wellbyt3

Summary

As the documentation mentions, the contracts are designed to be compatible with any EVM chain and support USDT:

> The smart contracts can potentially be implemented on any full-EVM chain
>
> USD is a generic name for a reference stable coin paired with BOOST in the AMO (USDC and USDT are the first natural candidates)

However, both the `V2AMO` and `V3AMO` contracts will not work with USDT, as they will revert during the `_addLiquidity()` and `_unfarmBuyBurn()` functions.

Root Cause

Both `V2AMO` and `V3AMO` use OpenZeppelin's IERC20Upgradeable interface, which expects a boolean return value when calling the `approve()` function. However, USDT's implementation of the approve() function does not return a boolean value, which causes the contract to revert during execution.

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
 ↪  of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {
```

The functions `_addLiquidity()` and `_unfarmBuyBurn()` in both contracts expect a boolean return value, causing them to revert when interacting with USDT.

example V3AMO::_addLiquidity and V3AMO::_unfarmBuyBurn:

```
 function _addLiquidity(uint256 usdAmount, uint256 minBoostSpend, uint256
 ↪  minUsdSpend, uint256 deadline)
        internal
        override
        returns (uint256 boostSpent, uint256 usdSpent, uint256 liquidity)
    {
        //....

        // Approve the transfer of BOOST and USD tokens to the pool
        IERC20Upgradeable(boost).approve(pool, boostAmount);

@>      IERC20Upgradeable(usd).approve(pool, usdAmount);

        (uint256 amount0Min, uint256 amount1Min) = sortAmounts(minBoostSpend,
 ↪  minUsdSpend);

        uint128 currentLiquidity = IV3Pool(pool).liquidity();
        liquidity = (usdAmount * currentLiquidity) /
 ↪  IERC20Upgradeable(usd).balanceOf(pool);

        // Add liquidity to the BOOST-USD pool within the specified tick range
        (uint256 amount0, uint256 amount1) = IV3Pool(pool).mint(
            address(this), tickLower, tickUpper, uint128(liquidity), amount0Min,
 ↪  amount1Min, deadline
        );

        // Revoke approval from the pool
        IERC20Upgradeable(boost).approve(pool, 0);
@>      IERC20Upgradeable(usd).approve(pool, 0);

      //....
    }

     function _unfarmBuyBurn(
        uint256 liquidity,
        uint256 minBoostRemove,
        uint256 minUsdRemove,
        uint256 minBoostAmountOut,
        uint256 deadline
    )
        internal
        override
        returns (uint256 boostRemoved, uint256 usdRemoved, uint256 usdAmountIn,
 ↪  uint256 boostAmountOut)
    {
       //....
        // Approve the transfer of usd tokens to the pool
@>      IERC20Upgradeable(usd).approve(pool, usdRemoved);

        // Execute the swap and store the amounts of tokens involved
        (int256 amount0, int256 amount1) = IV3Pool(pool).swap(
            address(this),
            boost > usd, // Determines if we are swapping USD for BOOST (true) or
 ↪  BOOST for USD (false)
```

```
            int256(usdRemoved),
            targetSqrtPriceX96,
            minBoostAmountOut,
            deadline
        );

        // Revoke approval from the pool
@>      IERC20Upgradeable(usd).approve(pool, 0);

        //...
    }
```

V2AMO::_addLiquidity and V2AMO::_unfarmBuyBurn faces the same issue

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

Adding liquidity and farming will fail due to a revert on USDT approvals

Mitigation

Use `safeApprove` instead of `approve`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/5

Issue M-3: Contracts of the codebase will not strictly compliant with the ERC-1504.

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/155

The protocol has acknowledged this issue.

Found by

0xnbvc, Atharv, AuditorPraise, HackTrace, Kirkeelee, dany.armstrong90, isagixyz, pkqs90

Summary

Contracts of the codebase isn't strictly compliant with the ERC-1504. This breaks the readme.

Root Cause

As per readme:

> Is the codebase expected to comply with any EIPs? Can there be/are there any deviations from the specification?
>
> Strictly compliant: ERC-1504: Upgradable Smart Contract

But the contracts of the codebase uses openzepplin upgradable contracts as base contract which are not compliant with ERC-1504. As per ERC-1504, the upgradable contract should consists of have handler contract, data contract and optionally the upgrader contract. But the contracts of the codebase are not compliant with ERC-1504 because they has no data contract and has data inside the handler contract.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

Break the readme.

PoC

No response

Mitigation

Make the contracts strictly compliant with ERC-1504.

Issue M-4: Precision loss in `boostPrice` in `V3AMO`

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/191

Found by

0x37, Bigsam, durov, s1ce, yuza101

Summary

There is precision loss in a specific case in `V3AMO`, which leads to incorrect value of `boostPrice` being reported. This leads to issues with upstream functions that use it.

Root Cause

https://github.com/sherlock-audit/2024-10-axion/blob/main/liquidity-amo/contracts/V3AMO.sol#L343

Consider the following code:

```
function boostPrice() public view override returns (uint256 price) {
        (uint160 _sqrtPriceX96, , , ) = IV3Pool(pool).slot0();
        uint256 sqrtPriceX96 = uint256(_sqrtPriceX96);
        if (boost < usd) {
            price = (10 ** (boostDecimals - usdDecimals + PRICE_DECIMALS) *
 ↪  sqrtPriceX96 ** 2) / Q96 ** 2;
        } else {
            if (sqrtPriceX96 >= Q96) {
                // @audit: massive precision loss here
                price = 10 ** (boostDecimals - usdDecimals + PRICE_DECIMALS) /
 ↪  (sqrtPriceX96 ** 2 / Q96 ** 2);
            } else {
                price = (10 ** (boostDecimals - usdDecimals + PRICE_DECIMALS) * Q96
 ↪  ** 2) / sqrtPriceX96 ** 2;
            }
        }
    }
```

Notice that in this specific clause:

```
if (sqrtPriceX96 >= Q96) {
                // @audit: massive precision loss here
                price = 10 ** (boostDecimals - usdDecimals + PRICE_DECIMALS) /
↪   (sqrtPriceX96 ** 2 / Q96 ** 2);
            }
```

We divide `(sqrtPriceX96**2/Q96**2)`. However, consider a case where the value of the stablecoin is 20% higher than the value of boost; in this case a price of around 0.8 should be reported by the function but because `(sqrtPriceX96**2/Q96**2)` will round down to 1, the function will end up reporting a price of 1.

Internal pre-conditions

No response

External pre-conditions

We need the price of the stablecoin to be at least a bit higher than the price of BOOST for this to be relevant

Attack Path

Described above; precision loss leads to boostPrice calculation reported by `V3AMO` being incorrect.

Impact

Impact is that, because the price is reported incorrect, public calls to `unfarmBuyBurn()` will fail because the following check will fail:

`if(newBoostPrice>boostUpperPriceBuy)revertPriceNotInRange(newBoostPrice);`

`newBoostPrice` will be reported as `1` even though it should be much lower.

PoC

First, set `sqrtPriceX96` to 87150978765690771352898345369 (10% above 2^96)

```
pool = await ethers.getContractAt("IV3Pool", poolAddress);
await pool.initialize(sqrtPriceX96);
```

Then:

```
it("Showcases the incorrect price that is returned", async function() {
        console.log(await V3AMO.boostPrice()) // 1000000
})
```

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/8

Issue M-5: MasterAMO should not use the `initializer` modifier

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/244

Found by

FonDevs, KupiaSec, UsmanAtique, calc1f4r, hunter_w3b

Summary

`MasterAMO` is a utils contract that intended to be inherited by V2AMO, V3AMO contracts, therefore. it's initializer function should not use the initializer modifier, instead, it should use onlyInitializing modifier.

Root Cause

In the MasterAMO.sol:104 contract, the initialize function uses the initializer modifier. This is incorrect for a contract like MasterAMO, which is meant to be inherited by other contracts, such as V2AMO and V3AMO.

In this inheritance model, the V2AMO contract also has its own initialize function, which includes the initializer modifier and calls the initialize function of MasterAMO. The problem here is that both the parent contract MasterAMO and the child contracts V2AMO, V3AMO are using the initializer modifier, which limits initialization to only one call.

According to the OpenZeppelin documentation, the onlyInitializing modifier should be used to allow initialization in both the parent and child contracts. The onlyInitializing modifier ensures that when the initialize function is called, any contracts in its inheritance chain can still complete their own initialization.

https://docs.openzeppelin.com/contracts/4.x/api/proxy#Initializable-initializer--

> A modifier that defines a protected initializer function that can be invoked at most once. In its scope, onlyInitializing functions can be used to initialize parent contracts.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

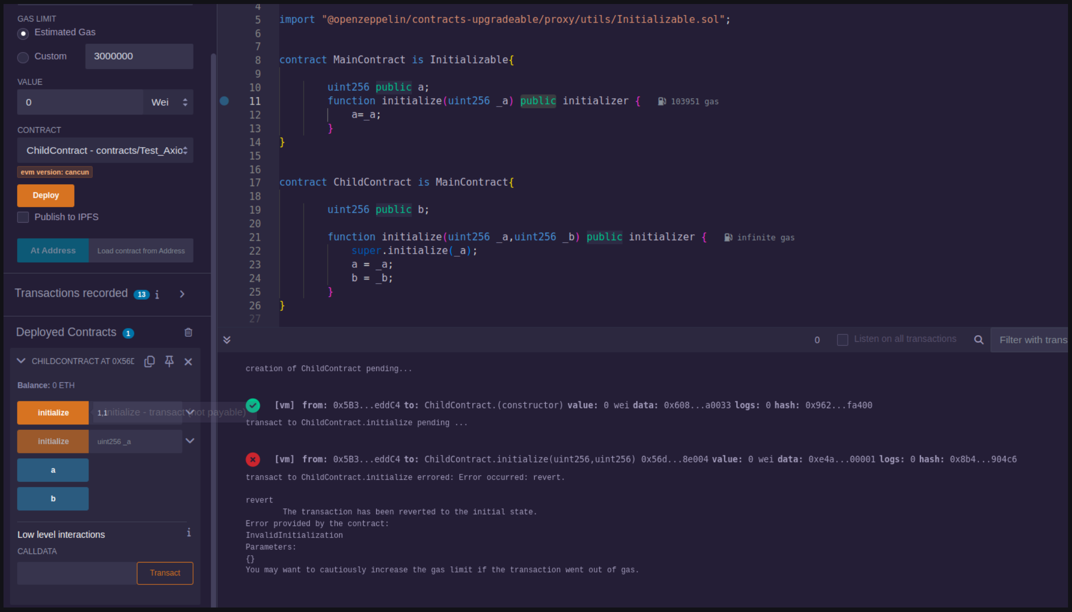
No response

Impact

In this scenario, no direct attack or monetary loss is likely. However, the vulnerability causes a significant operational issue, preventing inheriting contracts from completing initialization. This could lead to a failure in the deployment of critical protocol components, affecting the overall system functionality.

PoC

A simple POC in Remix.



```
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";


contract MainContract is Initializable{

        uint256 public a;
        function initialize(uint256 _a) public initializer {    103951 gas
            a=_a;
        }
}


contract ChildContract is MainContract{

        uint256 public b;

        function initialize(uint256 _a,uint256 _b) public initializer {    infinite gas
            super.initialize(_a);
            a = _a;
            b = _b;
        }
}
```

```
creation of ChildContract pending...

[vm]  from: 0x5B3...eddC4 to: ChildContract.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0x962...fa400
transact to ChildContract.initialize pending ...

[vm]  from: 0x5B3...eddC4 to: ChildContract.initialize(uint256,uint256) 0x56d...8e004 value: 0 wei data: 0xe4a...00001 logs: 0 hash: 0x8b4...904c6
transact to ChildContract.initialize errored: Error occurred: revert.

revert
        The transaction has been reverted to the initial state.
Error provided by the contract:
InvalidInitialization
Parameters:
{}
You may want to cautiously increase the gas limit if the transaction went out of gas.
```

Mitigation

Replace the initializer modifier in the MasterAMO contract with the onlyInitializing modifier. This allows the initialize function to be used by both the MasterAMO and any inheriting contracts during their initialization phase, without conflicting with their individual setup processes.

```
    function initialize(
        address admin, // // Address assigned the admin role (given exclusively to
↪   a multi-sig wallet)
        address boost_, // The Boost stablecoin address
        address usd_, // generic name for $1 collateral ( typically USDC or USDT )
        address pool_, // The pool where AMO logic applies for Boost-USD pair
        // On each chain where Boost is deployed, there will be a stable Boost-USD
↪   pool ensuring BOOST's peg.
```

```
        // Multiple Boost-USD pools can exist across different DEXes on the same
 ↪  chain, each with its own AMO, maintaining independent peg guarantees.
        address boostMinter_ // the minter contract
-   ) public initializer {
+   ) public onlyInitializing {
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/11

Issue M-6: The `V3AMO._mintAndSellBoost()` function does not work with Velodrome, Aerodrome, Fenix, Thena and Ramses

Source: https://github.com/sherlock-audit/2024-10-axion-judging/issues/268

Found by

ABA, Kirkeelee, KupiaSec, carrotsmuggler, pkqs90

Summary

The protocol mints `BOOST` tokens and sells them for USD using the `_mintAndSellBoost()` function for V3 DEXes. Since Velodrome, Aerodrome, Fenix, Thena and Ramses are parts of the V3 DEXes, they should be compatible. However, the `_mintAndSellBoost()` function does not work with the DEXes due to incorrect function parameters.

Root Cause

In the `_mintAndSellBoost()` function, it mints `BOOST` tokens and swaps them for USD tokens.

https://github.com/sherlock-audit/2024-10-axion/blob/00224d96daefa5ecc9a9795a97e144f7448a8e9e/liquidity-amo/contracts/V3AMO.sol#L144-L151

```
File: liquidity-amo\contracts\V3AMO.sol
144: @>      (int256 amount0, int256 amount1) = IV3Pool(pool).swap(
145:             address(this),
146:             boost < usd,
147:             int256(boostAmount), // Amount of BOOST tokens being swapped
148:             targetSqrtPriceX96, // The target square root price
149:             minUsdAmountOut, // Minimum acceptable amount of USD to receive
 ↪  from the swap
150:             deadline
151:         );
```

However, the `swap()` function of Velodrome, Aerodrome, Fenix, Thena and Ramses has different parameters. The `swap()` functions of the DEXes are as follows:

Velodrome : https://optimistic.etherscan.io/address/0xCc0bDDB707055e04e497aB22a59c2aF4391cd12F#code: :text=File%208%20of%2037%20%3A%20CLPool.sol L613-L619

Aerodrome : https://basescan.org/address/0x5e7BB104d84c7CB9B682AaC2F3d509f5F406809A#code: :text=File%208%20of%2037%20%3A%20CLPool.sol L679-L685

Fenix : https://blastscan.io/address/0x5aCCAc55f692Ae2F065CEdDF5924C8f6B53cDaa8#code: :text=File%202%20of%2044%20%3A%20AlgebraPool.sol L212-L218

Thena : https://bscscan.com/address/0xc89F69Baa3ff17a842AB2DE89E5Fc8a8e2cc7358#code: :text=File%202%20of%2031%20%3A%20AlgebraPool.sol L591-L597

Ramses : https://arbiscan.io/address/0xf896d16fa56a625802b6013f9f9202790ec69908#code: :text=File%2044%20of%2045%20%3A%20RamsesV2Pool.sol L944-L950

```
function swap(
  address recipient,
  bool zeroToOne,
  int256 amountRequired,
  uint160 limitSqrtPrice,
  bytes calldata data
) external override returns (int256 amount0, int256 amount1) {
```

As a result, the `_mintAndSellBoost()` function does not work with Velodrome, Aerodrome, Fenix, Thena and Ramses due to incorrect function parameters.

Internal pre-conditions

For convenience, let's assume that the USD token is `USDC` from this point forward.

- Protocol team is going to mint additional `BOOST` and sell them for `USDC` to bring the price back down to peg.

External pre-conditions

- The `BOOST-USDC` price diverges from peg and `BOOST` is trading above `$1` in Velodrome.

Attack Path

- Alice(protocol team) calls the `mintAndSellBoost()` function.

It reverts.

Impact

The `mintAndSellBoost()`, `mintSellFarm()` functions will be permanently DoSed for Velodrome, Aerodrome, Fenix, Thena and Ramses. Protocol team can't mint additional `BOOST` and sell them for `USDC` to bring the price back down to peg.

PoC

Mitigation

Use the correct function parameters for Velodrome, Aerodrome, Fenix, Thena and Ramses.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/AXION-MONEY/liquidity-amo/pull/13

Disclaimers