ALGEM LIQUID FARMING

SECURITY AUDIT

June 2025

Contents

Summary

ALgem Farming is a yield farming aggregator that allows users to deposit liquidity into vaults that farm on DEXes. Users receive dual rewards: farming rewards from the underlying DEX plus additional ALGM token incentives from the protocol, with progressive benefits for ALGM stakers.

Project: ALgem Farming

Website: www.algm.io

Audit Methodology

Manual review is the primary methodology used to identify vulnerabilities in smart contracts.

The objective of the audit is to increase security and provide solutions to resolve potential bugs that could undermine trust in a project.

Audit Scope

The contracts that are in the scope of this audit are the following:

Contract 1	Vault.sol
Contract 2	Pool.sol
Contract 3	Different Dapps

Vulnerability Founds

Findings:

5 High - Severity

4 Medium - Severity

3 Low - Severity

Explain Findings

High Severity

H - 01 KyoVaultV3 => deposit - Loss of funds and inflact all the calculations if small amounts of tokens are deposited into Vault.

Description:

It is possible to deposit small amounts of tokens in the protocol even if the amount of IWRAPPED tokens minted is 0.
This will lead to users losing funds because they cannot retrieve their tokens if the IWRAPPED balance is 0.
At the same time, this will inflate all calculations.

Details:

Users are allowed to deposit amounts of tokens in the vault to interact with a Uniswap V3 pool and get rewards.
Once a user deposits in the Vault, will get back half the amount of the value sent in WETH as IWRAPPED tokens.

```
function deposit(uint256 _amount) external payable
whenNotPaused updater {
      ...

    //half of user value is minted as LWRAPPED token
    uint256 wa = (amount1 +
getSecondAmount(uint128(amount0), true)) / 2;
   1.     p.wrapped += was;
     ...
}

```

There is no validation to ensure the minted lWRAPPED amount exceeds zero.
This creates a risk of fund loss for users, as those who receive zero lWRAPPED tokens cannot recover their deposits through the redeem or withdraw functions.
At the same time, all calculations could be inflated because the amount of tokens in the pool will increase while the lWRAPPED supply remains the same, leading to a mismatch in calculations.

PoC:

```
function test_depositRedeemSmallAmounts()public{

    vm.warp(block.timestamp + 86400);

    IERC20(pair).approve(address(v3vault), 1000 ether);
    v3vault.deposit{value: 1 ether}(1 ether);


    vm.startPrank(user1);
    IERC20(pair).approve(address(v3vault), 1000 ether);
    v3vault.deposit{value: 10000 wei}(10000 wei);
    vm.stopPrank();

    vm.warp(block.timestamp + v3vault.FINISH());

    vm.startPrank(user1);
    v3vault.redeem();
}
```

Impact:

Users lose funds and inflate calculations

Mitigation:

Add a check in deposit function that ensure the min amount of IWRAPPED tokens minted is more than 0.

Status:

Solved

H - 02 KyoVaultV3 => liquidate - Liquidation is impossible due to a logic error in liquidate function

Description:

The liquidate() function contains multiple critical arithmetic and logic errors that prevent all liquidations from executing successfully.
The function fails with arithmetic underflow/overflow panics, making the liquidation mechanism completely non-functional and breaking a core protocol safety feature.

Details:

The liquidation mechanism is designed to maintain the 1:1 ratio between WRAPPED and IWRAPPED tokens by liquidating under-collateralized positions when their Health Factor falls below LIQUIDATION_THRESHOLD.
However, the implementation contains several bugs:

```
function liquidate(address _user) external updater {
      ...

    (uint256 amount0, uint256 amount1) =
decreaseLiquidity(getUserLP(_user));
    (int256 amount0Delta,) = swap(-int256(p.wrapped -
amount1), true); amount0 -= uint256(amount0Delta);
   ...
 }
```

Issue 2.1 – Wrong Swap Direction:
When amount1 > p.wrapped (excess WETH received from liquidity), the calculation p.wrapped – amount1 becomes negative. The operation –int256(negative_value) becomes positive, causing the swap to attempt buying WETH instead of selling the excess.

Issue 2.2 – Arithmetic Underflow in uint256 Casting:
When amount0Delta is negative (which occurs during normal swap operations), uint256(amount0Delta) causes an arithmetic overflow, converting the negative value to a massive positive number.

Issue 2.3 – Subtraction Underflow:
The operation amount0 -= uint256(amount0Delta) with an overflowed positive value causes arithmetic underflow when subtracting from amount0.

Impact:

Complete Liquidation Mechanism Breakdown:
All liquidation attempts fail with panic errors.

Undercollateralized Positions Cannot Be Liquidated:
Positions with HF < threshold remain active.

Protocol Insolvency Risk:
Bad debt accumulation without liquidation mechanism.

1:1 WRAPPED/IWRAPPED Peg Failure:
Core protocol invariant cannot be maintained.

Mitigation:
The fix implements proper case handling for all possible scenarios during liquidation and uses safe arithmetic operations to prevent overflow/underflow errors.
Root Cause Resolution: The corrected code replaces the single-case assumption with comprehensive logic that handles three distinct scenarios:

1. Insufficient WETH case (amount1 < p.wrapped): When the liquidity withdrawal provides less WETH than required, the function correctly executes a buy swap to acquire the needed WETH amount.
2. Excess WETH case (amount1 > p.wrapped): When the liquidity withdrawal provides more WETH than needed, the function now properly executes a sell swap to convert the excess WETH into pair tokens.
3. Perfect match case (amount1 == p.wrapped): When the amounts match exactly, no swap is performed, eliminating unnecessary gas costs and slippage.

```
function liquidate(address _user) external updater {
    Position storage p = positions[_user];

    uint256 hf = getHF(_user);
    if (hf < LIQUIDATION_THRESHOLD) {
        _claim(_user, false);
        _unstakeALGM(_user, userALGMBalance[_user]);

        p.finish = block.timestamp;
        (uint256 amount0, uint256 amount1) =
decreaseLiquidity(getUserLP(_user))

        if (amount1 < p.wrapped) {
            console.log("Need to buy more WETH");
            (int256 amount0Delta,) = swap(-int256(p.wrapped -
amount1), true);
            amount0 = uint256(int256(amount0) + amount0Delta);
        } else if (amount1 > p.wrapped) {
            (int256 amount0Delta,) = swap(int256(amount1 -
p.wrapped), false);
            amount0 = uint256(int256(amount0) + amount0Delta);
        }

        totalTokenBalance += amount0;
        p.token = amount0;
        totalWRAPPEDBalance += p.wrapped;

        emit Liquidated(_user, p.wrapped, amount0);
    }
}
```

This approach correctly handles both positive and negative amount0Delta values returned by Uniswap V3 swaps, where negative values indicate tokens received and positive values indicate tokens owed.

Additional Security Enhancements: It is also recommended to implement reentrancy protection (H3 bug will show the reentrancy using the old/new code for liquidation) and liquidation incentives for external actors.

Status:

Solved

H - 03 KyoVaultV3 => liquidate - Reentrancy attack in liquidate function.

Description:

There is no reentrancy check on the liquidate function.
This will lead to a reentrancy attack on the liquidate function.

Details:

```
function liquidate(address _user) external updater {
    Position storage p = positions[_user];

    uint256 hf = getHF(_user);
    if (hf < LIQUIDATION_THRESHOLD) {
        _claim(_user, false);
        _unstakeALGM(_user, userALGMBalance[_user]);

        p.finish = block.timestamp;
        (uint256 amount0, uint256 amount1) =
decreaseLiquidity(getUserLP(_user))

        ...
    }
}
```

As it is possible to check from the liquidation function, there is a call to the _claim() function that performs some calculations and sends back the native tokens to the user in liquidation.

Many operations in the liquidate functions are performed after the _claim() call.

Supposing the user is a contract, there is the possibility of calling liquidation repeatedly using the receive() function.

This means that a user can call the liquidate function recursively, liquidating always the same position.

Impact:

- **Fund drainage:** Multiple liquidations allow extraction of more ETH/tokens than deserved
- **State corruption:** Variables totalWRAPPEDBalance, supp, positions get manipulated
- **Protocol collapse:** Potential vault insolvency

Mitigation:

Add nonReentrant modifier.

Status:

Solved

PoC:

```
function test_liquidateReenter()public{
    vm.warp(block.timestamp + 86400);

    IERC20(pair).approve(address(v3vault), 1000 ether);
    v3vault.deposit{value: 1 ether}(1 ether);

    vm.startPrank(user1);
    IERC20(pair).approve(address(v3vault), 1000 ether);
    v3vault.deposit{value: 2 ether}(2 ether);

    vm.warp(block.timestamp + 86400);

    v3vault.liquidate(address(this));
    vm.stopPrank();
}



receive() external payable {
    console.log("IN RECEIVE FUNCTION");
    v3vault.liquidate(address(this));
    }
}

```

H - 04 KyoVaultV3 => withdraw, redeem, liquidate

-These functions suffer logic problems that can lead to insolvency for protocol.

Description:

Some price movements from the market can lead the protocol to insolvency, because the amounts of ETH from the swaps can be lower than the amount set as p.wrapped from the user when depositing.

Details:

Liquidate() function

The liquidate() function is triggered when the value of a user's collateral falls below the liquidation threshold required by the protocol.

At this point, when liquidation is called, it performs a swap directly to the pool and gets an amount of ETH.

The amount of ETH received from this swap should be the p.wrapped amount of the user that is set when a user initially deposits into the protocol.

If the market falls, and a user goes into liquidation, the amount from the swap() call could be lower than the actual p.wrapped, leading to insolvency because the liquidated user can't redeem the real amount obtained and then can't use the protocol anymore due to the logic of the contract that doesn't permit a new deposit if the user hasn't redeemed the position.

Withdraw () function

The withdraw() function is called when the value expires and has the same problem as liquidate above.

If the market falls and the withdrawal function is callable, it can lead to insolvency due to the same logic as liquidation.

Extra:
The withdraw() function lacks the update to the supp variable, which leads to incorrect calculations for getUserLP, which can be used for redemption

Redeem() function

The redeem() function has different execution branches.

If the market falls and redeem() is called, it can lead to insolvency due to the same underlying logic flaw present in both liquidate() and withdraw() functions.

Impact:

Insolvency.
Risk of wrong calculations in other parts of the contracts.
Exclude users from using the protocol.
User fund lock.

PoC:

```
function test_ExtremePriceManipulation() public {
        console.log("=== TEST: Extreme Price Manipulation ===");

        vm.warp(v3vault.START() + 1000);

        uint256 input = 1 ether;
        deposit(input, user1);

        vm.startPrank(user3);
        IERC20(pair).approve(address(v3vault), 1000 ether);
        v3vault.deposit{value: 5 ether}(5 ether);
        vm.stopPrank();

        vm.warp(v3vault.START() + 86400);

        // === Extreme Price Manipulation ===
        priceImpact(user1);

        // vm.warp(block.timestamp + v3vault.FINISH());

        // ==== LIQUIDATION TEST AFTER PRICE IMPACT ======
        //v3vault.liquidate(user1);


        //// ==== REDEEM TEST AFTER PRICE IMPACT ======
        vm.startPrank(user1);
        v3vault.redeem();
        vm.stopPrank();

        //// ==== WITHDRAW TEST AFTER PRICE IMPACT ======
        //vm.startPrank(user1);
        //v3vault.withdraw(IERC20(lwrapped).balanceOf(user1));
        //vm.stopPrank();
    }
```

Mitigation:

Redesign the underlying logic to prevent protocol insolvency.

Check every single variable that is part of the calculations:
supp
totalWrappedBalance
...

Status:

Solved

From Protocol:

"The core idea of the protocol is to supply minted LWRAPPEDs with ETH. If market falls really deep in a moment, we will not be able to supply it anyways until the price settled back. Our workaround here is the liquidation script mechanism, which is checking for such conditions on a hourly rate and liquidates those risky positions before the insolvency occurred. Other approach implies that we would burn LWRAPPED in other than 1:1 ratio which is not suitable for the core idea."
...

H-05 KyoVaultV3 =>Logic error in liquidate() function, let the HF stable during time.

Description:

The liquidation mechanism fails to protect the protocol during price manipulation attacks because Health Factor calculations rely on TWAP prices instead of real-time spot prices, allowing underwater positions to avoid liquidation.

Details:

The liquidation system contains a fundamental design flaw where Health Factor calculations use time-averaged prices that don't reflect current market conditions:

```
function liquidate(address _user) external nonReentrant
updater {
        Position storage p = positions[_user];
        uint256 hf = getHF(_user);
        if (hf < LIQUIDATION_THRESHOLD) {
            ...
        }
    }
```

The getHF() call during the flow calls other 2 functions in the contract get a HF result:

- calculateRemoveLiquidity()
- getSecondAmount()

These two functions internally use the TWAP price of the pool to determine the HF of a user, but at the same time, the TWAP price reflects an average of 1 hour.

This means that if someone manipulates NOW the pool, the HF will be stable, because the TWAP price doesn't recognize the actual manipulation.

Impact

A user that should be liquidate still get the HF above the LIQUIDATION_THRESHOLD.

PoC

A user that should be liquidate still get the HF above the LIQUIDATION_THRESHOLD.

```
function test_Liquidations() public {
        console.log("=== TEST: Liquidation ===");

        vm.warp(v3vault.START() + 86400);

        vm.startPrank(user6);
        vm.warp(block.timestamp + 50);
        vm.roll(block.number + 3);
        IERC20(pair).approve(address(v3vault), 1000 ether);
        v3vault.deposit{value: 0.1 ether}(100 ether);
        vm.stopPrank();

        console.log("getHF user6 Before", v3vault.getHF(user6));
        priceImpactSendETH(user6, 1000 ether);
        console.log("getHF user6 After", v3vault.getHF(user6));

        vm.startPrank(user7);
        v3vault.liquidate(user6);
        vm.stopPrank();
    }
```

In the provided test, we see exactly this issue:

- Multiple large swaps dramatically change spot price
- Health Factor remains constant at 199% despite a price movements
- Liquidation never triggers despite significant price manipulation

Mitigation:

The liquidate function should get the HF based on the slot0 prices to reflect the real-time changes.

Status:

Solved

Medium Severity

M- 01 VelodromeV3 =>setTWAPparams - Missing function to setTWAPparams.

Description:

The setTWAPparams() function is missing in the Velodrome vault.

Mitigation:

Add the same setTWAPparams() function that is used in the other dapps.

Status:

Solved

M- 02 KyoVaultV3 =>Triple call on calculate removeLiquidity() function in the deposit() function could lead to inconsistency in calculation

- Redundant calls in the deposit functions

Description:

Calling the calculateRemoveLiquidity three times in the deposit function could lead to inconsistent calculations.

Details:

The Deposit() function calls internally three times the calculateRemoveLiquidity() function and this can create inconsistency between values that are passed then to the other calls of the deposit.

```
function deposit(uint256 _amount) external payable
whenNotPaused updater {
    uint256 cr = getCurrentRound();
    Position storage p = positions[msg.sender];
    _checkIfCanDeposit(_amount, msg.value, p, cr);

    if (msg.value > 0) {
        IWETH9(WRAPPED).deposit{value: msg.value}();
    }
    if (_amount > 0) {
        IERC20(pairToken).safeTransferFrom(msg.sender,
address(this), _amount);
    }

    uint256 cratio = getCurrentRatio();

    uint256 a0;
    uint256 a1;
    if (cratio < targetRatio - targetRatio / 10 || cratio >
targetRatio + targetRatio / 10) {

        uint128 tb = uint128(totalBalance());
        uint256[2] memory crl = calculateRemoveLiquidity(tb);

        uint256 nratio = getNewRatio(crl[0] + _amount +
dustLeft[0], crl[1] + msg.value + dustLeft[1]);

        nratio = cratio < targetRatio
            ? nratio > targetRatio ? targetRatio : nratio
            : nratio < targetRatio ? targetRatio : nratio;

        (a0, a1) = decreaseLiquidity(tb);

        (positionParameters.tickL, positionParameters.tickU) =
getNewTicks(nratio);
    }
```

As it is possible to check, the deposit() function calls the calculateRemoveLiquidity in:

- getCurrentRatio()
- calculateRemoveLiquidity()
- getNewRatio

This can create inconsistency between calculations in the flow of deposit() function that could leads to wrong calculations.

Impact

Wrong calculations can be perform.

Mitigation:

Re-think the logic behind the deposit function to use consistency values between the calls and prevent wrong calculations.

Status:

Solved

M - 03 KyoVaultV3 => getPrice, getTick, getBounds - These functions are based on the pool.slot0() calculations that could be easily manipulated.

Description:

A Market Manipulation Attack can be performed on the pool to impact the calculations that are made by these 3 functions.

Details:

```
function getPrice() internal view returns (uint160 price) {
    (price,,,,,,) = v3pool.slot0();
    console.log("price from get price", price);
}

function getTick() internal view returns (int24 tick) {
    (, tick,,,,,) = v3pool.slot0();
}

function getBounds() external view returns (uint256,
uint256, uint256, bool) {
    (, int24 tick,,,,,) = v3pool.slot0();
    uint256 pl = getPriceAtTick(positionParameters.tickL);
    uint256 pc = getPriceAtTick(tick);
    uint256 pu = getPriceAtTick(positionParameters.tickU);
    return (pl, pc, pu, pc >= pl && pc <= pu);
}
```

The data pulled from Uniswap.slot0, which is the most recent data point, can be easily manipulated via MEV bots and Flashloans with sandwich attacks, potentially causing the loss of funds when interacting with Uniswap.swap function.
This could lead to wrong calculations and loss of funds for the protocol and other users

Impact:

- **Wrong calculations:** Every protocol interaction can be easily manipulated from anyone.
- **Protocol collapse:** Potential vault insolvency

Mitigation:

Use the TWAP price and deviation check, which will be compared to slot0 to get a reasonable sqrtPriceX96.

TWAP is a pricing algorithm used to calculate the average price of an asset over a set period.
It is calculated by summing prices at multiple points across a set period and then dividing this total by the total number of price points

Status:

Solved

M-04 KyoVaultV3 =>calculateAddLiquidity - Add and compare the TWAP price to the slot0 price and add a deviation check.

Description:

The protocol lacks price deviation checks between TWAP and spot prices in liquidity calculations, enabling price manipulation attacks during deposits.

Details:

The `calculateAddLiquidity` function uses spot price from `slot0()` without validating against TWAP, allowing attackers to exploit temporary price manipulations:

```
function calculateAddLiquidity(uint256 _amount0, uint256
_amount1) public view returns (uint128 liquidity_) {

        uint160 sqrtRatioX96 = getPrice();
        //(, uint160 TWAPsqrtRatioX96) = getTWData();
        uint160 sqrtRatioAX96 =
        ....
            );
        }
    }
```

At this point, the manipulation that can be made are differents and you can refer to the **M-03** bug.

The correct way to prevents a manipulation could be the compare between the slot0 and TWAP price and calculate a deviation or a series of deviations that could be accepted.

```
function calculateAddLiquidity(uint256 _amount0, uint256
_amount1) public view returns (uint128 liquidity_) {

    uint160 sqrtRatioX96 = getPrice();
    (, uint160 TWAPsqrtRatioX96) = getTWData();

    uint256 deviation = sqrtRatioX96 > TWAPsqrtRatioX96
        ? ((sqrtRatioX96 - TWAPsqrtRatioX96) * 10000) /
TWAPsqrtRatioX96
        : ((TWAPsqrtRatioX96 - sqrtRatioX96) * 10000) /
TWAPsqrtRatioX96;

    if (deviation > 500 && deviation < 1000) {
        .....
    }
    else{
     .....
     }
;
```

The logic behind this check is simple.
Limit/prevents situations where the prices are inflated and could leads to loss of funds for users.

The Price deviation thresholds should vary by asset volatility:

- Stablecoins (USDC/USDT): 1-2% max deviation acceptable
- Volatile pairs (ETH/ALT): 5-10% max deviation acceptable
- Exotic pairs: Higher thresholds may be required

Implement graduated protection with deviation-based controls:

These examples can be used as a demonstration of the logic suggested to limit/prevents manipulations from bad actors that could lead to loss of funds for users.

The same logic could be added even to the calculateRemoveLiquidity but using more permissive parameters.

Impact

Manipulated prices could be used.

Mitigation:

Implement graduated protection with deviation-based controls.

Status:

Solved

Low Severity

L - 01 V3Caller.sol => increaseLiquidity - Unused calculations.

Description:

The calculateRemoveLiquidity function is called but its return value is never used within the function scope.

Mitigation:

Remove the calculateRemoveLiquidity call from the function to eliminate unnecessary calls.

Status:

Solved

L - 02 All the contracts missing a gap, for future implementations

Description:

Missing storage gap for future implementations in all upgradeable contracts

Mitigation:

Add a storage gap *(e.g., uint256[50] private __gap;)* at the end of each upgradeable contract to reserve storage slots for future variables.

Status:

Solved

L - 03 Vault.sol => setALGMStaking - ALGM Staking Contract can't be set more than once.

Description:

If the ALGM staking contract encounters issues, the Vault lacks functionality to update the contract reference. This forces a complete redeployment of all contracts rather than simply updating the staking address.

Mitigation:

Remove the
require(address(algmStaking) == address(0));
check, in the setALGMStaking() function

Status:

Solved

Conclusion

Post-analysis review confirms that all critical and non-critical issues identified in the smart contracts have been successfully addressed.
The implemented solutions ensure protocol integrity and provide robust protection against potential security threats.

Disclaimer

I analyzed these smart contracts following industry best practices current at the date of this report, focusing on vulnerabilities and issues within the smart contract source code as detailed in this report.

it is crucial to understand that this report should not serve as your sole security assessment.

I strongly recommend implementing a bug bounty program to validate the security level of these smart contracts.

I bear no responsibility for fund safety and accepts no liability for project security.
Smart contracts operate on blockchain platforms, and the underlying platform, programming languages, and related software may contain inherent vulnerabilities that could enable attacks.

Therefore, this audit cannot provide absolute security guarantees for the audited smart contracts. Users and stakeholders assume full responsibility for any risks associated with the deployment and use of these contracts.