// Security Assessment

11.18.2025 - 11.21.2025

Decentralized Borrowing Protocol

Liquid Loans

HALBORN

Decentralized Borrowing Protocol - Liquid Loans

Prepared by: HALBORN

Last Updated 12/12/2025

Date of Engagement: November 18th, 2025 - November 21st, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	1	0	0	1	1

TABLE OF CONTENTS

1. INTRODUCTION

`Liquid Loans` engaged Halborn to conduct a security assessment on their smart contracts beginning on November 18th, 2025 and ending on November 20th, 2025. The security assessment was scoped to the smart contracts provided in the Github repository, provided to the Halborn team. Commit hash and further details can be found in the Scope section of this report.

The reviewed contracts **LoanAirdrop** and **CommunityIssuance** contracts work together to distribute LOAN tokens to eligible users through a merkle-based airdrop and protocol-driven issuance schedule. The **LoanAirdrop** contract validates user allocations via Merkle proofs and releases tokens over predefined monthly claim windows, while the **CommunityIssuance** contract mints and streams LOAN rewards to stability pool participants based on protocol conditions. These components rely on price-oracle caller contracts, which fetch TWAP prices from Uniswap pools to support accurate valuation and reward calculations, ensuring that token distribution and issuance remain aligned with real-time market conditions.

2. ASSESSMENT SUMMARY

`Halborn` was provided with 3 days for this engagement and assigned a full-time security engineer to assess the security of the smart contracts in scope. The assigned engineer possess deep expertise in blockchain and smart contract security, including hands-on experience with multiple blockchain protocols.

The objective of this assessment is to:

- Identify potential security issues within the `Liquid Loans` protocol smart contracts.
- Ensure that smart contract of `Liquid Loans` protocol functions operate as intended.

In summary, `Halborn` identified several areas for improvement to reduce the likelihood and impact of security risks, which were partially addressed by the `Liquid Loans team`. The main ones were:

- `Update the inversion logic to correctly rescale based on token decimals.`
- `Upgrade the contract to Newer Solidity version.`
- `Upgrade all OpenZeppelin dependencies to the latest stable version.`

3. TEST APPROACH AND METHODOLOGY

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture and purpose of the `Liquid Loans` protocol.
- Manual code review and walkthrough of the `Liquid Loans` in-scope contracts.
- Manual assessment of critical `Solidity` variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).
- Local deployment and testing with (`Foundry` , `Remix IDE`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORIES

(a) Repository: monorepo

(b) Assessed Commit ID: 765ec6d

(c) Items in scope:

- packages/contracts/contracts/LOAN/LockupAirdrop.sol
- packages/contracts/contracts/LOAN/CommunityIssuance.sol

Out-of-Scope: Third party dependencies and economic attacks.

(a) Repository: oracle-callers

(b) Assessed Commit ID: 28acf2b

(c) Items in scope:

- contracts/UniswapV2Caller.sol
- contracts/UniswapV3Caller.sol
- contracts/SecondaryCallerUpdateable.sol
- contracts/libraries/TickMathWrapper.sol
- contracts/interfaces/IERC20Metadata.sol
- contracts/interfaces/IPriceFeedSecondaryUpdateable.sol
- contracts/interfaces/ITickMathWrapper.sol
- contracts/interfaces/IUniswapV2Pool.sol
- contracts/interfaces/IUniswapV3Pool.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- https://github.com/Liquid-Loans-Official/oracle-callers/pull/30/commits/ffd8f911c3a117c860052b123c7bb7b7ac43fa9cd
- https://github.com/Liquid-Loans-Official/monorepo/tree/audit

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW

1 0 0 1

INFORMATIONAL
1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT DECIMAL NORMALIZATION IN UNISWAP V2 ORACLE PRICE CALCULATIONS LEADING TO PRICE INFLATION	CRITICAL	SOLVED - 11/24/2025
USE OF OUTDATED SOLIDITY VERSION	LOW	RISK ACCEPTED - 11/24/2025
USE OF OUTDATED OPENZEPPELIN LIBRARIES	INFORMATIONAL	ACKNOWLEDGED - 11/24/2025

7. FINDINGS & TECH DETAILS

7.1 INCORRECT DECIMAL NORMALIZATION IN UNISWAP V2 ORACLE PRICE CALCULATIONS LEADING TO PRICE INFLATION

// CRITICAL

Description

In the `UniswapV2Caller` contract, the `update()` function normalizes prices as if both tokens always use 18 decimals. This leads to severe inflation when pools involve tokens with fewer decimals (e.g., USDC with 6 decimals). The root vulnerability is missing decimal normalization before scaling to `1e18`, causing `~1e12` over-scaling in ETH/USD pricing.

In `UniswapV2Caller` :
The V2 oracle takes Uniswap's `UQ112x112` cumulative prices and directly converts them into `1e18` precision:

```
44 | lastPriceX18 = (avgUQ112x112 * 1e18) >> 112;
```

Copy Code

This implicitly assumes both ***tokens*** use 18 decimals. In a USDC/ETH pool (token0 = USDC, 6 decimals), the conversion multiplies the price by **`1e12`**, resulting in values such as **~324,408,010 ETH per USDC** instead of the expected **~0.0003 ETH per USDC**. This breaks any protocol consuming ETH/USD or USD-denominated feeds, leading to corrupted collateralization logic, incorrect liquidations, and oracle-driven insolvency risk.

Due to this:

- Liquity-style oracle consumers receive ETH prices inflated by `1e12`.
- Collateralization ratios become meaningless.
- Liquidations may trigger incorrectly.
- Lending systems become insolvent or reject valid operations.
- Any economic logic depending on accurate ETH/USD price becomes corrupted.

Proof of Concept

Place the `UniswapV2CallerPoC.t.sol` file inside the `foundry/test` directory. This proof-of-concept highlights a **decimal-inversion bug** in the `UniswapV2Caller` contract: when the oracle is configured with `invert = true` , the TWAP returned from Uniswap V2 is inverted but **not re-scaled to the correct decimal format**, causing a silent **mispricing** (USDC decimals ignored). By forking Ethereum mainnet and comparing the real ETH price against the oracle's output, the test demonstrates that the oracle reports a

price that is off by decimals, proving that inverted TWAP values must be normalized to token decimals before converting from UQ112x112 to 1e18 format.

Copy Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";

interface IERC20 {
    function decimals() external view returns (uint8);
    function balanceOf(address) external view returns (uint256);
}

interface IUniswapV2Pool {
    function token0() external view returns (address);
    function token1() external view returns (address);
    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim
    function price0CumulativeLast() external view returns (uint256);
    function price1CumulativeLast() external view returns (uint256);
}

interface IPriceFeedSecondaryUpdateable {
    function getLastValue() external view returns (bool success, uint256 value, uint256 timestamp, by
    function getCurrentValue() external returns (bool success, uint256 value, uint256 timestamp, byte
}

contract UniswapV2Caller is IPriceFeedSecondaryUpdateable {

    IUniswapV2Pool public immutable pair;
    bool public immutable invert;

    uint32 public immutable minWindow;
    uint32  public timestampLast;
    uint256 public lastPriceX18;
    uint256 public priceCumulativeLast;

    constructor(address _pair, uint32 _minWindow, bool _invert) {
        require(_pair != address(0), "zero pair");
        require(_minWindow > 0, "bad window");

        pair = IUniswapV2Pool(_pair);
        invert = _invert;
        minWindow = _minWindow;

        // initialize snapshot
        (, , uint32 ts) = pair.getReserves();
        timestampLast = ts;
        priceCumulativeLast = invert ? pair.price0CumulativeLast() : pair.price1CumulativeLast();
    }

    function update() public {
        uint256 cumulativeNow = invert ? pair.price0CumulativeLast() : pair.price1CumulativeLast();
        (, , uint32 ts) = pair.getReserves();

        uint32 elapsed = ts - timestampLast;

        // Enforce minimum window
        if (elapsed < minWindow) {
            // Not enough time: just return last stored price
            return;
        }

        // BUG: No decimal normalization!
        uint256 avgUQ112x112 = (cumulativeNow - priceCumulativeLast) / elapsed;
        lastPriceX18 = (avgUQ112x112 * 1e18) >> 112;

        priceCumulativeLast = cumulativeNow;
        timestampLast = ts;
    }

    function getLastValue()
        public
```

```
        view
        override
        returns (bool success, uint256 value, uint256 timestamp, bytes32 data)
    {
        if (lastPriceX18 == 0) {
            return (false, 0, 0, bytes32(0));
        }
        return (true, lastPriceX18, timestampLast, bytes32(0));
    }

    function getCurrentValue()
        external
        override
        returns (bool success, uint256 value, uint256 timestamp, bytes32 data)
    {
        update();
        return getLastValue();
    }
}

contract UniswapV2CallerPoC is Test {
    // Real Ethereum mainnet addresses
    address constant USDC_ETH_POOL = 0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc;
    address constant USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
    address constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    IUniswapV2Pool public pool;

    function setUp() public {
        console.log("=== Pool Configuration ===");
        console.log("Pool:", USDC_ETH_POOL);
        console.log("USDC:", USDC);
        console.log("WETH:", WETH);
    }

    function test1DecimalBugDemonstration() public {
        console.log("\n=== DEMONSTRATING DECIMAL BUG ON REAL POOL ===\n");

        // Fork at a specific block
        vm.createSelectFork("https://eth-mainnet.g.alchemy.com/v2/TJYxxlJAsVXGQS15KRdJaTIhtM00iWzF",

        pool = IUniswapV2Pool(USDC_ETH_POOL);

        // Verify token order
        address token0 = pool.token0();
        address token1 = pool.token1();

        console.log("Token0:", token0, "(USDC)");
        console.log("Token1:", token1, "(WETH)");
        console.log("USDC decimals:", IERC20(USDC).decimals());
        console.log("WETH decimals:", IERC20(WETH).decimals());

        // Get initial state
        (uint112 reserve0_initial, uint112 reserve1_initial, uint32 ts_initial) = pool.getReserves();
        uint256 price0Cumulative_initial = pool.price0CumulativeLast();

        console.log("\n--- Initial State (Block 21233000) ---");
        console.log("Reserve0 (USDC):", reserve0_initial);
        console.log("Reserve1 (WETH):", reserve1_initial);
        console.log("Timestamp:", ts_initial);
        console.log("price1CumulativeLast:", price0Cumulative_initial);

        // Calculate real ETH price
        uint256 realPriceUSD_initial = (uint256(reserve0_initial) * 1e18) / uint256(reserve1_initial)
        console.log("REAL ETH price: $", realPriceUSD_initial / 1e18);

        // Create oracle with invert=false (reads price1 = ETH price in USDC)
        UniswapV2Caller oracleBuggy = new UniswapV2Caller(USDC_ETH_POOL, 60, true);
        console.log("Oracle deployed at:", address(oracleBuggy));
        console.log("Oracle initialized with timestampLast:", oracleBuggy.timestampLast());

        // Now we need to advance to a block where pool was updated
        // We'll roll forward on the SAME fork to a later block
        console.log("\n=== Advancing to Later Block ===");
        vm.rollFork(21233010); // Roll forward 10 blocks (~120 seconds)

        // Get state at new block
        (uint112 reserve0_later, uint112 reserve1_later, uint32 ts_later) = pool.getReserves();
```

```
        uint256 price0Cumulative_later = pool.price0CumulativeLast();

        console.log("\n--- Later State (Block 21233010) ---");
        console.log("Reserve0 (USDC):", reserve0_later);
        console.log("Reserve1 (WETH):", reserve1_later);
        console.log("Timestamp:", ts_later);
        console.log("price1CumulativeLast:", price0Cumulative_later);
        console.log("Time elapsed:", uint256(ts_later) - uint256(ts_initial), "seconds");
        console.log("Price cumulative change:", price0Cumulative_later - price0Cumulative_initial);

        // Calculate real ETH price
        uint256 realPriceUSD_later = (uint256(reserve0_later) * 1e18) / uint256(reserve1_later);
        console.log("REAL ETH price: $", realPriceUSD_later / 1e18);

        // Check if enough time passed
        if (ts_later <= ts_initial) {
            console.log("\n!!! WARNING: Pool timestamp didn't advance !!!");
            console.log("This means no swaps occurred between these blocks");
            console.log("Try different block numbers or use vm.warp");
            revert("Pool timestamp didn't advance");
        }

        if ((ts_later - ts_initial) < 60) {
            console.log("\n!!! WARNING: Less than 60 seconds elapsed !!!");
            console.log("Oracle requires minWindow of 60 seconds");
            revert("Insufficient time elapsed");
        }

        console.log("\n=== Updating Oracle ===");

        // Trigger update
        oracleBuggy.update();

        console.log("Oracle lastPriceX18:", oracleBuggy.lastPriceX18());
        console.log("Oracle timestampLast:", oracleBuggy.timestampLast());
        console.log("Oracle priceCumulativeLast:", oracleBuggy.priceCumulativeLast());

        // Get oracle price
        (bool success, uint256 oraclePrice, uint256 oracleTimestamp, ) = oracleBuggy.getLastValue();

        console.log("\n=== ORACLE OUTPUT (BUGGY) ===");
        console.log("Success:", success);
        console.log("Oracle timestamp:", oracleTimestamp);
        console.log("Oracle price (raw):", oraclePrice);

        require(success, "Oracle should return success");
        require(oraclePrice > 0, "Oracle price should be non-zero");


        // Assertions
        assertTrue(success, "Oracle should return success");
        assertTrue(oraclePrice > 0, "Oracle should return non-zero price");

    }

}
```

The test passes, confirming that the UniswapV2Caller oracle suffers from a severe decimal-normalization bug. In this PoC, 1 USDC should have resolved to approximately `0.0003246 ETH`, but instead the oracle reports an impossible price of `324408010.060800616278278536 ETH` per USDC.

```
Ran 1 test for test/LastFinalPOC2.t.sol:UniswapV2CallerPoC
[PASS] test1DecimalBugDemonstration() (gas: 670596)
Logs:
  === Pool Configuration ===
  Pool: 0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc
  USDC: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48
  WETH: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2

=== DEMONSTRATING DECIMAL BUG ON REAL POOL ===

  Token0: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48 (USDC)
  Token1: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2 (WETH)
  USDC decimals: 6
  WETH decimals: 18

--- Initial State (Block 21233000) ---
  Reserve0 (USDC): 43689084181807
  Reserve1 (WETH): 14172589575625284352000
  Timestamp: 1732154591
  price1CumulativeLast: 637685580021696650510053863592321324938315425942808
  REAL ETH price: $ 0
  Oracle deployed at: 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
  Oracle initialized with timestampLast: 1732154591

=== Advancing to Later Block ===

--- Later State (Block 21233010) ---
  Reserve0 (USDC): 43683088249308
  Reserve1 (WETH): 14174541048609960395392
  Timestamp: 1732154687
  price1CumulativeLast: 637685741726275036644114237667651174478053056713520
  Time elapsed: 96 seconds
  Price cumulative change: 161704578386134060374075329849539737630770712
  REAL ETH price: $ 0

=== Updating Oracle ===
  Oracle lastPriceX18: 324408010060800616278278536
  Oracle timestampLast: 1732154687
  Oracle priceCumulativeLast: 637685741726275036644114237667651174478053056713520

=== ORACLE OUTPUT (BUGGY) ===
  Success: true
  Oracle timestamp: 1732154687
  Oracle price (raw): 324408010060800616278278536

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 742.05ms (738.82ms CPU time)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:H/D:H/Y:N (10.0)

Recommendation

It is recommended to apply proper token decimal normalization for V2 calculations before scaling to 1e18, ensuring the final price reflects correct 18-decimal precision regardless of pool decimals.

Remediation Comment

SOLVED: The **Liquid Loans team** solved the issue by applying token decimal normalization for V2 calculations.

Remediation Hash

https://github.com/Liquid-Loans-Official/oracle-callers/pull/30/commits/ffd8f911c3a117c86052b123c7bb7b7ac43fa9cd

7.2 USE OF OUTDATED SOLIDITY VERSION

// LOW

Description

The contract uses an outdated Solidity compiler version:

```
pragma solidity 0.6.11;
```

Solidity `0.6.x` is more than four years old and lacks numerous safety features, compiler optimizations, and built-in protections introduced in later versions (especially 0.8.x). Using such an old compiler introduces several risks:

1. **Missing Overflow/Underflow Protection**
   Versions prior to `0.8.0` do not include automatic arithmetic safety checks, causing potential silent overflows unless manually handled.
2. **Missing language improvements**
   Features like custom errors, immutable variables, receive/fallback improvements, safer ABI encoding, and memory optimizations are unavailable.
3. **Potential incompatibility with modern dependencies**
   Most modern libraries (OpenZeppelin etc.) have dropped support for Solidity <0.7, increasing maintenance burden.

Because this contract interacts with token transfers and Merkle-based claims, using an older compiler increases the risks of undefined behavior and developer mistakes.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Upgrade the contract to Solidity `^0.8.20`.

Remediation Comment

RISK ACCEPTED: The **Liquid Loans team** accepted the risk of this finding.

Remediation Hash

https://github.com/Liquid-Loans-Official/monorepo/tree/audit

7.3 USE OF OUTDATED OPENZEPPELIN LIBRARIES

// INFORMATIONAL

Description

The contract imports OpenZeppelin contracts from an older **v3.4-era** codebase:

```
Copy Code
4 import "../Dependencies/Ownable.sol";
5 import "../Dependencies/MerkleProof.sol";
6 import "../Dependencies/SafeERC20.sol";
7 import "../Dependencies/ReentrancyGuard.sol";
```

These versions were designed for Solidity **0.6.x**, lack many modern safety improvements, and are no longer maintained. Relying on outdated versions introduces several risks:

1. **Missing modern security patches**
2. **Outdated access control (Ownable)**
3. **Older `MerkleProof` implementation**
4. **Legacy `SafeERC20` behavior**
5. **Older `ReentrancyGuard`**
6. **Incompatibility with modern toolchains**

Using these older dependencies increases the likelihood of subtle bugs, reduces interoperability, and limits auditability.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

Upgrade all OpenZeppelin dependencies to the latest stable version.

Remediation Comment

ACKNOWLEDGED: The **Liquid Loans team** acknowledged this finding.

Remediation Hash

https://github.com/Liquid-Loans-Official/monorepo/tree/audit
