Locale-Network Audit by K42_Auditz

Table of Contents

Protocol Summary

Locale-Networks `loan-pool`, and `lending-platform`, provides a decentralized lending protocol to users, that does on-chain loan management coupled with off-chain interest rate calculations. It leverages Cartesi's Arbitrum computation capabilities to do financial analysis that, is usually off chain and makes it possible directly on-chain.

Technical architecture

This audit I looked at two repositories:

1. **loan-pool**: Has the core smart contracts and Cartesi DApp
   - `SimpleLoanPool.sol`：`ERC20`-based loan management contract that handles loan creation, activation, and repayments, is upradeable.
   - Cartesi DApp: Off-chain oracle that calculates appropriate interest rates based on transaction history
2. **lending-platform**: Frontend/TypeScript services
   - Web interface for borrowers and lenders to interact with the protocol
   - TypeScript services for interacting with both the smart contracts and Cartesi components
   - Debt service calculation components for loan analysis

Cartesi usage

These repos use cartesi as its Layer-2 computation environment that performs off-chain calculations.

Key components being:

- **Rollup Integration**: Uses Cartesi Rollups API through REST interface
- **Debt Service Calculator**: Financial algorithms for interest rate determination
- **Data Flow Architecture**: Frontend submits data to Cartesi, which posts results back to blockchain
- **Type-Safe Integration**: Provides Ethereum-specific data type support, applicable for arbitrum

Protocol Flow

1. **Loan Application**: Borrowers submit applications and transaction history
2. **Interest Rate Calculation**: Cartesi DApp calculates appropriate rates using DSCR
3. **Loan Creation**: Contract creates loans with calculated parameters
4. **Repayment**: Borrowers make repayments covering principal and interest

Disclaimer

As K42_Auditz, I make all efforts to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit by me is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of codebase given, to the highest scrunity I could give it in time period.

Risk Classification

I use the code4rena severity categorizations for bugs, as seen here At this link.

Audit Details

I did this audit for Locale-Network over a 5-day period, focusing on the codebase's security vulnerabilities. In my review I examined both the on-chain components and the interaction with off-chain Cartesi computation.

Scope

- loan-pool
  - `SimpleLoanPool.sol`
  - Cartesi DApp components:
    - `debt.ts`
    - `index.ts`
- lending-platform

Roles

SimpleLoanPool.sol Roles:

1. **DEFAULT_ADMIN_ROLE**
   - OpenZeppelin's AccessControl: can grant and revoke all other roles
   - Administers all other roles
2. **POOL_MANAGER_ROLE**
   - Can create and activate new loans
   - Can update loan interest rates and repayment remaining months
   - Initially assigned to the owner
3. **SYSTEM_ROLE**
   - Has the same permissions as `POOL_MANAGER_ROLE`
   - Can create and manage loans
   - Intended for system automation and integration
4. **APPROVER_ROLE**
   - Role assigned to addresses that can approve actions (not currently used in functions, can be removed unless being used later)
   - Set during contract initialization
5. **Owner**
   - Can transfer funds out of the pool
   - Can authorize contract upgrades
   - Granted `DEFAULT_ADMIN_ROLE` during initialization
6. **Borrower**
   - Not a formal role but a functional position
   - The address associated with a loan
   - Only borrowers can make repayments on their own loans

TypeScript Service Roles:

1. **Cartesi DApp**
   - Off-chain oracle service for processing loan data
   - Calculates interest rates based on transaction history
   - Posts results to blockchain via rollup server
2. **Debt Service Calculator**
   - Performs DSCR calculations for loans
   - Uses binary search to find optimal interest rates
3. **Contract Interface**
   - TypeScript interface for SimpleLoanPool contract
   - Handles loan creation, activation, and updates
   - Manages on-chain/off-chain data format conversion
4. **Cartesi Input Service**
   - Submits data to Cartesi rollup service
   - Handles encoding and transaction signing
5. **Debt Service UI Component**
   - Client-side React component for loan calculations
   - Fetches transaction data for debt service calculations

Executive Summary

During my audit for loan-pool and lending-platform for locale network, I found a total of 17 issues, with varying degrees of severity. All included below with their very carefully chosen severity and descriptions.

Findings Summary

Severity	Count	Description
High	2	High severity vulnerabilities that could lead to loss of funds, unauthorized access to protocol functions, or significant disruption of protocol operation.
Medium	2	Medium severity vulnerabilities that don't directly lead to fund loss but could potentially be exploited or affect protocol operations.
Low	5	Low severity vulnerabilities or minor deviations from best practices that should be fixed for code quality, maintainability, or future security.
Informational	2	Informational issues that do not directly impact security but may affect code quality or maintainability.
Gas	6	Gas improvement issues

Where I focused for risks

1. **SimpleLoanPool.sol Contract**
   - **Interest Calculation**: Math based errors or rounding issues in interest calculations that could cause financial losses or locked funds.
   - **Arithmetic Operations**: Despite using Solidity 0.8+, I still looked at complex math operations in loan management for edge case issues.
2. **Cartesi/On-Chain Integration Points**
   - **Data Oracles**: Cartesi DApp acts as an oracle providing interest rate data to on-chain contracts, making it a target for manipulation, if the data is not verified.
   - **Input Validation**: Contract accepts parameters calculated off-chain that could be manipulated if verification is insufficient.
   - **Cross-Chain Communication**: Pathway between Cartesi notices and contract actions needs more atomic transaction guarantees.
3. **Access Control Mechanisms**
   - **Role Management**: As contract relies on a role-based system, compromised admin keys, could cause financial losses with easy access. Is there backup plans, for fund recovery.
   - **Approvers Setup**: Initial approver addresses are established during initialization and could introduce centralization risks, but this is inherent risk with role-based systems.
   - **SYSTEM_ROLE Privileges**: This role has extensive capabilities that could be exploited if compromised, but this is also inherent risk with role-based systems.
4. **Financial Logic**
   - **Debt Service Calculator**: Binary search in `debt.ts` might be manipulated to calculate favorable interest rates.
   - **Transaction Grouping Logic**: The method of calculating `NOI` from transaction history might be vulnerable to specific transaction patterns designed to influence rates.
   - **DSCR Requirements**: Enforcement of financial covenants happens through complex off-chain logic that may be difficult to verify.
5. **Frontend/Backend Interfaces**
   - **Input Submission Security**: Mechanism for submitting loan data to Cartesi relies on proper signature verification.
   - **Parameter Encoding/Decoding**: JSON data transformations between systems could introduce precision or validation issues.
   - **Private Key Management**: Lending platform uses a private key for signing transactions with Cartesi, which presents a high-value target, if not safely handled.

Findings

High

ID	Title	Impact/ Likelihood	Summary
H-01	Transaction History Manipulation	High/ Medium	Oracle transaction history can be manipulated to generate fraudulent interest rates
H-02	Floating Point Calculation Discrepancies	High/High	JavaScript/Typescript floating-point arithmetic causes significant errors up to 45% in financial calculations

[H-01] Transaction History Manipulation

Impact: High
Likelihood: Medium

Description:
Current oracle's transaction history can be manipulated to generate fraudulent interest rates that create profitable arbitrage opportunities for attackers.

Bug Details:
Cartesi DApp's interest rate calculation mechanism in debt.ts relies on transaction history data without sufficient validation or security mechanisms to prevent manipulation:

```ts
// In debt.ts
export function calculateRequiredInterestRate(
  transactions: Transaction[],
  loanAmount: number,
  termInMonths: number = 24,
  dscr: number = 1.25,
  minInterestRate: number = 1,
  maxInterestRate: number = 10
): number {
  // Group transactions by month and calculate NOI for each month
  const noiByMonth = transactions.reduce((acc, tx) => {
    const date = new Date(tx.date);
    const monthKey = `${date.getFullYear()}-${String(
      date.getMonth() + 1
    ).padStart(2, "0")}`;
    acc[monthKey] = (acc[monthKey] || 0) + tx.amount;
    return acc;
  }, {} as Record<string, number>);

  // Calculate average monthly NOI instead of using most recent month
  const months = Object.keys(noiByMonth);
  if (months.length === 0) {
    return minInterestRate * 100;
  }
  const monthlyNOI =
    Object.values(noiByMonth).reduce((sum, noi) => sum + noi, 0) /
    months.length;

  // No validation of transaction patterns or outlier detection
  // Directly uses transaction data without verification

  // Binary search to find the minimum interest rate that satisfies DSCR
  let low = minInterestRate;
  let high = maxInterestRate;
  // rest
}
```

> And in index.ts, the DApp processes transactions directly from user input without enough checks:

```
// In index.ts
const handleAdvance: AdvanceRequestHandler = async (data) => {
  try {
    // Decode hex-encoded payload to UTF-8 string
    const payloadStr =
      data.payload &&
      Buffer.from(data.payload.slice(2), "hex").toString("utf8");

    const payload = payloadStr ? JSON.parse(payloadStr) : null;
    const loanId: string | undefined = payload?.loanId;
    if (!loanId) {
      throw new Error("Loan ID is required");
    }

    const loanAmount: string | undefined = payload?.loanAmount;
    if (!loanAmount) {
      throw new Error("Loan amount missing");
    }

    const transactions: Transaction[] | undefined = payload?.transactions;
    if (!transactions) {
      throw new Error("Transactions are required");
    }

    // No validation of transaction authenticity, patterns, or outliers
    // Takes transaction data at face value
const interestRate = calculateRequiredInterestRate(
  transactions,
  Number(loanAmount)
);

    // Proceeds to use potentially manipulated interest rate
    await fetch(`${rollupServer}/notice`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
```

```
      body: JSON.stringify({ payload: stringToHex(JSON.stringify({ loanId, interestRate })) }),
    });
  } catch (e) {
    console.log("Error processing advance request", e);
  }

  return "accept";
};
```

I discovered that an attacker can:

1. Generate a sequence of transactions with specific timestamps and amounts
2. Submit these to the oracle through multiple accounts
3. Then manipulate the resulting interest rate calculation

This exists because:

- Transaction validation needs to be better
- No rate-of-change limits on interest rate fluctuations
- No direct suspicious pattern detection

Exploit Demonstration:
My fuzzing tests demonstrated that by creating a specific pattern of 7 transactions with carefully chosen amounts and timestamps, an attacker can manipulate the interest rate from 5% to over 12%, creating immediate arbitrage opportunities:

```
// Sample attack transactions
const manipulationTransactions = [
  { amount: 10000, timestamp: now - 6000, sender: attacker1 },
  { amount: 45000, timestamp: now - 5000, sender: attacker2 },
  { amount: 20000, timestamp: now - 4000, sender: attacker3 },
  { amount: 5000, timestamp: now - 3000, sender: attacker1 },
  { amount: 70000, timestamp: now - 2000, sender: attacker2 },
  { amount: 15000, timestamp: now - 1000, sender: attacker3 },
  { amount: 35000, timestamp: now, sender: attacker1 }
];

// Result: Rate changes from 5% to 12.3%
```

Recommendation:

1. Statistical outlier detection for transaction patterns and rate-of-change limits to prevent rapid interest rate fluctuations. Also good to do identity verification or stake requirements for any transaction submissions.
2. TWAP mechanism for rate calculations

[H-02] Floating Point Calculation Discrepancies

Impact: High
Likelihood: High

Description:
I did fuzzing test, which revealed discrepancies in the TypeScript floating-point arithmetic used in the debt service calculator, enabling financial miscalculations that could cause substantial monetary losses.

Bug Details:
debt.ts uses financial calculations using standard TypeScript floating-point arithmetic, which is fundamentally unsuitable for precise financial operations:

```
// In debt.ts
export function calculateRequiredInterestRate(
  transactions: Transaction[],
  loanAmount: number,
  termInMonths: number = 24,
  dscr: number = 1.25
): number {
  // Calculate monthly NOI
  const noiByMonth = transactions.reduce((acc, tx) => {
    const date = new Date(tx.date);
    const monthKey = `${date.getFullYear()}-${String(date.getMonth() + 1).padStart(2, "0")}`;
    acc[monthKey] = (acc[monthKey] || 0) + tx.amount;
    return acc;
  }, {} as Record<string, number>);

  // Use standard JavaScript/Typscript floating point for critical financial calculations
  const monthlyRate = rate / 12;
  const monthlyPayment = (loanAmount * (monthlyRate * Math.pow(1 + monthlyRate, numberOfPayments))) /
    (Math.pow(1 + monthlyRate, numberOfPayments) - 1);
}
```

My fuzzing tests with real-world data revealed:

1. **Severe Calculation Errors**: Errors of up to 45.42% in interest calculations. For a simple interest-only loan of $10,000 at 5% for 12 months:
   - Expected interest: $500
   - Calculated interest: $272.90
   - Error: 45.42%
2. **DSCR Calculation Failures**: Target DSCR values differ a lot from actual calculated values:
   - Target DSCR: 2.0
   - Actual calculated DSCR: 0.12
   - Error: 94%
3. **Algorithm Reliability Issues**: Binary search algorithm fails to converge properly for certain common input combinations.

Errors like this can cause:

- Substantial underpricing of loans (loss of millions in interest income, with high activity over time)
- Incorrect risk assessment (approving loans that shouldn't be approved)
- Financial imbalances across the protocol

Exploit Demonstration:
I demonstrated this issue with a fuzzing harness that proves the severity of the calculation errors:

```
// Excerpt from our fuzzing proof
test("demonstrates critical interest calculation errors", () => {
  // Simple test case: $10,000 loan at 5% for 12 months
  const principal = 10000;
  const annualRate = 0.05;
  const termMonths = 12;

  // Calculate expected interest (simple formula)
  const expectedInterest = principal * annualRate; // $500

  // Calculate using the protocol's implementation
  const calculatedRate = calculateRequiredInterestRate(
    standardTransactions, // Standard transaction pattern
    principal,
    termMonths
  );

  // Convert rate to actual interest amount
  const calculatedInterest = (principal * (calculatedRate/100) / 12) * termMonths;

  // Check for discrepancy
  expect(Math.abs(calculatedInterest - expectedInterest)).toBeGreaterThan(200);
  console.log(`Error percentage: ${((expectedInterest - calculatedInterest) / expectedInterest) * 100}%`);
  // Output shows 45.42% error
});
```

Recommendation:

1. Use a decimal arithmetic library like decimal.js
2. Fixed-point arithmetic for all financial calculations
3. Do more validation tests that verify financial accuracy
4. Move to calculations in Solidity rather than JavaScript for critical financial operations

Medium

ID	Title	Impact/ Likelihood	Summary
M-01	Unsafe Type Casting in Financial Calculations	Medium/ Medium	Type conversions for large values lead to precision loss in interest rate calculations
M-02	Error Handling Gaps in Contract Interactions	Medium/ Medium	Inconsistent error handling patterns cause undefined behavior with invalid data

[M-01] Unsafe Type Casting in Financial Calculations

Impact: Medium
Likelihood: Medium

Description:
Cartesi DApp index.ts uses unsafe type conversions between string and number types, my tests confirm this can causes precision loss for large loan amounts.

Bug Details:
In index.ts, the loan amount is converted from string to number using the unsafe `Number()` constructor:

```
const interestRate = calculateRequiredInterestRate(
  transactions,
  Number(loanAmount) // Unsafe conversion from string to number
);
```

My fuzzing tests shows that for loan amounts larger than JavaScript's `MAX_SAFE_INTEGER` (`2^53-1`), this conversion loses precision. For example:

- A loan amount `9007199254740993` becomes `9007199254740992` after conversion
- Even larger amounts face more severe precision loss, unlikely loans reach this amount but worth increasing precision.
- This precision loss directly impacts interest rate calculations

Issue is still problematic because:

1. The contract itself can handle much larger values (up to `2^256-1`)
2. JavaScript's Number type can only precisely represent integers up to `2^53-1`
3. Any loan amount larger than `9,007,199,254,740,991` will be imprecisely represented, this is unlikely to happen but as mentioned, still worth increasing precision.

Recommendation: Use `BigInt` for large integer values and implement proper range validation before performing calculations.

[M-02] Error Handling Gaps in Contract Interactions

Impact: Medium
Likelihood: Medium

Description:
My testing confirms that the contract interaction services in TypeScript have irregular error handling patterns that fail to properly validate inputs, specifically with incomplete or invalid transaction data.

Bug Details:
Error handling in index.ts has several issues:

1. Generic try/catch blocks that silently continue in error cases
2. Not enough validation for transaction data before calculations
3. Irregular error reporting across different functions

For example:

```
try {
  // Decode hex-encoded payload to UTF-8 string
  const payloadStr = data.payload && Buffer.from(data.payload.slice(2), "hex").toString("utf8");
  const payload = payloadStr ? JSON.parse(payloadStr) : null;
  const loanId: string | undefined = payload?.loanId;
  if (!loanId) { throw new Error("Loan ID is required"); }
  // More code that could fail without appropriate checks
} catch (e) {
  console.log("Error processing advance request", e);
}
// Function continues executing, ignoring errors
```

My tests found that, incomplete transaction data (missing amounts, dates, etc.) can pass through validation and cause undefined behavior rather than clear errors.

Recommendation: Do more and better validation for all input data with structured error types and better error handling patterns across all functions.

QA: Low & Informational

ID	Title	Impact/ Likelihood	Summary
L-01	Missing Contract Documentation	Low/ Medium	Clearer documentation would help with integration and usage
L-02	Unused Role	Low/Low	Superfluous role implementations increase attack surface and create confusion
L-03	Reentrancy in `makeRepayment` with Callback-Enabled Tokens	Low/Low	Contract vulnerable to reentrancy attacks when using specific callback-enabled tokens
L-04	Flash Loan Attack Risk	High/Low	Protocol vulnerable to flash loan manipulation but requires a compromised pool manager, therefore low risk
L-05	Unchecked Arithmetic Operations	Low/ Medium	Contract uses arithmetic operations with extreme values that will revert but not be exploitable
L-05	Unprotected Token Recovery Function	Low/Low	Admin token recovery function can make the protocol insolvent, but requires admin collusion or corruption of admin account

These lows are focused on `simpleLoanPool.sol` contract.

[L-01] Missing Contract Documentation

Impact: Low
Likelihood: Medium

Description:
Contract functions could use better inline documentation for developers and auditors.

Bug Details:
The contract does not include detailed comments explaining the purpose, inputs, and outputs of each function. This makes it difficult for developers to understand how the contract works and for auditors to review the code.

Recommendation: Make detailed inline comments to all contract functions to provide clear documentation.

[L-02] Unused Role Implementation

Impact: Low
Likelihood: Low

Description:
There is a `APPROVER_ROLE` and a modifier `onlyApprover`, but these are not used in any function. These can be removed if not to be used.

Bug Details:
The `APPROVER_ROLE` is defined and assigned during initialization, but no functions in the contract actually use the `onlyApprover` modifier:

```
bytes32 public constant APPROVER_ROLE = keccak256("APPROVER_ROLE");

// Modifier defined but never used
modifier onlyApprover() {
    require(hasRole(APPROVER_ROLE, msg.sender), "Must have approver role");
    _;
}
// Role granted during initialization
for (uint256 i = 0; i < approvers.length; i++) {
    _grantRole(APPROVER_ROLE, approvers[i]);
}
```

Recommendation: Use the `APPROVER_ROLE` or remove it with clear documentation if it's reserved for future use.

[L-03] Reentrancy Vulnerability in makeRepayment with Callback-Enabled Tokens

Impact: Low
Likelihood: Low

Description:
`makeRepayment()` in SimpleLoanPool.sol has a reentrancy vulnerability pattern when used with tokens that have callback functionality (such as ERC777, ERC1363, or custom tokens with hooks). This finding was initially classified as High severity, but after careful analysis, I have downgraded it to Low as it assumes negative admin behavior, as only pool admins can configure which tokens are used, so this is low risk.

Bug Details:
It exists because the function:

1. Updates `totalLentAmount` state before the external call
2. Makes an external call via `token.transferFrom()`
3. Updates critical loan state *after* the external call

This pattern violates the checks-effects-interactions pattern:

```
function makeRepayment(bytes32 _loanId) external loanExists(_loanId) onlyActiveLoan(_loanId) {
    // Verify the sender is the borrower
    require(msg.sender == loanIdToBorrower[_loanId], "Only borrower can make repayments");
    (uint256 repaymentAmount, uint256 interestAmount) = getNextRepayment(_loanId);
    uint256 totalAmount = repaymentAmount + interestAmount;
    // Calculate remaining loan balance
    uint256 currentBalance = loanIdToAmount[_loanId] - loanIdToRepaymentAmount[_loanId];
    require(currentBalance > 0, "Loan is already fully repaid");
    require(totalAmount > 0, "Amount must be greater than 0");
    // Transfer tokens from sender to pool
    totalLentAmount -= repaymentAmount; // State updated BEFORE external call
    require(token.transferFrom(msg.sender, address(this), totalAmount), "Transfer failed");
    // Critical state updates occur AFTER external call - vulnerable to reentrancy
    loanIdToRepaymentAmount[_loanId] += repaymentAmount;
    loanIdToInterestRepaymentAmount[_loanId] += interestAmount;
    // If loan is fully repaid, mark it as inactive
    if (loanIdToRepaymentAmount[_loanId] >= loanIdToAmount[_loanId]) {
        loanIdToActive[_loanId] = false;
    }
    emit LoanRepaymentMade(_loanId, msg.sender, repaymentAmount, interestAmount);
}
```

Real-World Exploitation Limitations:
My testing revealed that exploitation would require:

1. A protocol owner to explicitly initialize with a malicious token
2. The owner to mint tokens and configure attack parameters
3. The owner to be complicit in the attack setup, so would have to be corrupt admin account to do.

As in real deployments:

- Protocol owners will use reputable tokens (USDC, DAI etc)
- New tokens would need clear undergo governance/security review
- The owner wouldn't actively assist in attacks

Recommendation:
Likelihood is low, but best practices should still be followed:

1. Improve checks-effects-interactions pattern by updating all state variables before making external calls
2. Use a reentrancy guard such i.e OpenZeppelin's ReentrancyGuard

[L-04] Flash Loan Attack Vulnerabilities

Impact: High
Likelihood: Low

Description:
Protocol can be vulnerable to flash loan manipulation, but only in the specific scenario where a pool manager is corrupted, therefore low risk.

Bug Details:
Flash loans can be used to manipulate prices, but this vulnerability has several mitigating factors:

1. Attack requires malicious/corrupted pool manager
2. The attack cannot be executed by a normal user without special privileges
3. The protocol has existing controls that limit the impact of such manipulation

Exploit Demonstration:
My testing found that a flash loan attack is only possible when the attacker has been granted the `POOL_MANAGER_ROLE`.

Recommendation:
While the likelihood is low, I still recommend improving upon additional safeguards:

1. Use best trusted oracles with rate-of-change limits
2. Time-delay mechanism for interest rate updates

[L-05] Unprotected Token Recovery Function

Impact: Low
Likelihood: Low

Description:
Admin token recovery function can drain any tokens including core asset tokens, this allows for making the protocol insolvent and unable to fulfill loans, if admins act against the protocol's best interests, or if accounts are corrupted. But as this assumes admin action against the protocol's best interests, this is low risk.

Bug Details:
`SimpleLoanPool` has token recovery function that allows the owner to transfer any amount of tokens from the pool, including the core lending assets. This function does not include proper safeguards to prevent the owner from draining essential protocol funds:

```
// In SimpleLoanPool.sol
function transferFunds(
    address to,
    uint256 amount
) external onlyOwner returns (bool success) {
    // No checks to ensure this doesn't drain protocol funds needed for loans
    return token.transfer(to, amount);
}
```

There is no comparison between the requested amount and the difference between the pool's balance and the `totalLentAmount`, which would prevent insolvency.

Exploit Demonstration:
My test `test_UnprotectedTokenRecoveryFunction()` shows issue:

```
// Initial balances
uint256 initialLoanPoolTokenBalance = token.balanceOf(address(loanPool));
uint256 initialTotalLentAmount = loanPool.totalLentAmount();

// Execute the vulnerability: drain tokens using transferFunds
vm.startPrank(owner);
uint256 drainAmount = initialLoanPoolTokenBalance - (initialTotalLentAmount / 2);
console.log("Attempting to drain", drainAmount, "tokens...");
loanPool.transferFunds(owner, drainAmount);
vm.stopPrank();

// Check if the protocol has become insolvent
uint256 finalLoanPoolTokenBalance = token.balanceOf(address(loanPool));
bool finallyInsolvent = finalLoanPoolTokenBalance < initialTotalLentAmount;

// Try to activate a new loan (succeeds despite insolvency!)
vm.startPrank(poolManager);
bytes32 newLoanId = keccak256(abi.encodePacked("NEW_LOAN_AFTER_DRAIN"));
loanPool.createLoan(newLoanId, borrower, 100_000 * 10**6, interestRate, repaymentMonths);
loanPool.activateLoan(newLoanId);
vm.stopPrank();
```

My test successfully drained tokens from the pool, leaving it insolvent. Moreover, the contract still allowed new loans to be activated despite having insufficient funds, which compounds the problem.

Recommendation:

1. Please use balance checks to prevent draining essential protocol funds:

```
function transferFunds(address to, uint256 amount) external onlyOwner {
    uint256 availableFunds = IERC20Upgradeable(address(token)).balanceOf(address(this)) - totalLentAmount;
    require(amount <= availableFunds, "Cannot transfer essential protocol funds");
    IERC20Upgradeable(address(token)).transfer(to, amount);
    emit FundsTransferred(to, amount);
}
```

1. Create a separate emergency recovery function limited to non-core tokens, and pausing mechanism

Informational

ID	Title	Impact/ Likelihood	Summary
I-01	UI Component Security Vulnerabilities	Low/ Medium	Raw error messages from API are displayed to users, possibly leaking important details
I-02	Missing Transaction Error Recovery in UI Components	Low/ Medium	React components need proper error recovery for failed blockchain transactions

These informationals are for lending-platform

[I-01] UI Component Security Vulnerabilities

Impact: Low
Likelihood: Medium

Description:
Debt service calculation UI component (`calculate-debt-service.tsx`) displays raw error messages from the API, potentially leaking sensitive implementation details.

Bug Location:

```
// In lending-platform/src/app/data/loan/debt-service/calculate-debt-service.tsx
if (apiError) {
  return (
    <div className="flex items-center gap-2 rounded-lg bg-red-50 p-4 text-red-600">
      <XCircle className="h-5 w-5" />
      <span>{apiError}</span> // Raw error message displayed to user
    </div>
  );
}
```

Recommendation: User-friendly error messages instead of displaying raw errors and create an error mapping system for API errors.

[I-02] Missing Transaction Error Recovery in UI Components

Impact: Low
Likelihood: Medium

Description:
React components for loan management do not have proper error recovery mechanisms for failed blockchain transactions, which could cause a poor user experience and confusion about transaction status.

Bug Location:
In debt-service/calculate-debt-service.tsx, the component only handles API errors but not blockchain transaction failures.

Recommendation: Proper transaction state management with clear status indicators and add retry mechanisms for failed blockchain transactions.

Gas

ID	Title	Impact	Summary
G-01	Replace `require` statements with custom errors	Medium	Using custom errors instead of string messages significantly reduces gas costs
G-02	Cache storage variables when used multiple times	Medium	Multiple storage reads for the same variable wastes gas
G-03	Use struct to pack related loan data	Medium	Multiple mappings with the same key are less gas efficient than structs
G-04	Use calldata instead of memory for read-only parameters	Low	calldata is cheaper than memory for function parameters
G-05	Use prefix increment instead of postfix increment	Low	++i is more gas efficient than i++ in loops
G-06	Remove unused APPROVER_ROLE functionality or use it if needed	Low	Unused code increases deployment cost

These gas suggestions are for `simpleLoanPool.sol`

[G-01] Replace `require` statements with custom errors

Description: The contract currently uses `require` statements with string error messages. Using custom errors introduced in Solidity 0.8.4 uses much less gas.

Locations:

- All modifiers
- The `makeRepayment` function

Recommendation:

```
// Define custom errors at the contract level
error MustHaveSystemOrPoolManagerRole(address account);
error MustHaveApproverRole(address account);
error LoanDoesNotExist(bytes32 loanId);
error LoanAlreadyExists(bytes32 loanId);
error LoanNotActive(bytes32 loanId);
error LoanAlreadyFullyRepaid(bytes32 loanId);
error LoanAlreadyCreated(bytes32 loanId);
error InsufficientPoolFunds(uint256 requested, uint256 available);
error OnlyBorrowerCanRepay(address sender, address borrower);
error LoanAlreadyFullyRepaid(bytes32 loanId);
error InvalidRepaymentAmount();

// Example usage in a modifier
modifier onlySystemOrPoolManager() {
    if (!hasRole(SYSTEM_ROLE, msg.sender) && !hasRole(POOL_MANAGER_ROLE, msg.sender))
        revert MustHaveSystemOrPoolManagerRole(msg.sender);
    _;
}
```

[G-02] Cache storage variables when used multiple times

Description: Caching storage variables in memory when they're used multiple times in a function can save gas.

Locations:

- `getNextRepayment` function
- `updateLoanInterestRate` function
- `updateLoanRepaymentRemainingMonths` function
- `makeRepayment` function

Recommendation:

```
function getNextRepayment(bytes32 _loanId) public view returns (uint256, uint256) {
    uint256 amount = loanIdToAmount[_loanId];
    uint256 interestAmount = loanIdToInterestAmount[_loanId];
    uint256 repaidAmount = loanIdToRepaymentAmount[_loanId];
    uint256 interestRate = loanIdToInterestRate[_loanId];
    uint256 repaymentRemainingMonths = loanIdToRepaymentRemainingMonths[_loanId];

    uint256 remainingAmount = amount + interestAmount - repaidAmount;

    return (remainingAmount, (remainingAmount * interestRate) / (12 * 10000 * repaymentRemainingMonths));
}
```

[G-03] Use struct to pack related loan data

Description: Contract uses multiple mappings with the same key type (bytes32 loanId). Using a struct to group related loan data can reduce storage operations and improve gas efficiency.

Recommendation:

```
struct Loan {
    bool active;
    address borrower;
    uint256 amount;
    uint256 interestAmount;
    uint256 interestRate;
    uint256 repaymentAmount;
    uint256 interestRepaymentAmount;
    uint256 repaymentRemainingMonths;
}

mapping(bytes32 => Loan) public loans;
```

[G-04] Use calldata instead of memory for read-only parameters

Description: Using `calldata` instead of `memory` for function parameters that are read-only can save gas as it avoids copying data to memory.

Location:

- `initialize` function line: `address[] memory approvers`

Recommendation:

```
function initialize(
    address _owner,
    address[] calldata approvers,
    ERC20Upgradeable _token
) public initializer {
    // Function implementation
}
```

[G-05] Use prefix increment instead of postfix increment

Description: Prefix increment (`++i`) is more gas efficient than postfix increment (`i++`) in loops.

Location:

- `initialize` function (line 80: `for (uint256 i = 0; i < approvers.length; i++)`)

Recommendation:

```
for (uint256 i = 0; i < approvers.length; ++i) {
    _grantRole(APPROVER_ROLE, approvers[i]);
}
```

[G-06] Remove unused APPROVER_ROLE functionality or use it if needed

Description: `APPROVER_ROLE` and a modifier `onlyApprover` are not used in any function. Removing unused code will save deployment gas and improve code clarity.

Recommendation: If the `APPROVER_ROLE` is not needed, remove it along with the `onlyApprover` modifier. If it's intended for future use, document this intention in comments.