ThunderLoan Audit Report

Version 1.0

Lokapal

February 3, 2025

ThunderLoan Audit Report

Lokapal

February 3, 2025

Prepared by:

- Lokapal

Lead Auditor:

- Ricardo Pintos

Table of Contents

    * [H-2] `ThunderLoan::flashloan` function only checks for returning the money to the protocol, which can be met by calling `deposit` instead of `repay`
    * [H-3] Mixing up variable location when upgrading contract causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`
  - Medium
    * [M-1] Using `TSwap` as a price oracle enables Price and Oracle Manipulation Attacks by reduced fees
    * [M-2] `ThunderLoan::initialize` can be called by anyone after deployment, potentially disrupting protocol functionality
  - Low
    * [L-1] Reentrancy in `ThunderLoan::flashloan`
    * [L-2] State variable changes in `ThunderLoan::updateFlashLoanFee` but no event is emitted
    * [L-3] `ThunderLoan__ExhangeRateCanOnlyIncrease` custom error not used
    * [L-4] Centralization risk for trusted owners
    * [L-5] Missing checks for `address(0)` in `OracleUpgradeable::__Oracle_init_unchained`
    * [L-6] `public` functions not used internally could be marked `external`
    * [L-7] Events with missing `indexed` fields
    * [L-8] PUSH0 is not supported by all chains
  - Informational
    * [I-1] The test coverage of the protocol is too low
    * [I-2] `ThunderLoan::setAllowedToken` is restricted to the owner, increasing centralization
    * [I-3] `AssetToken::updateExchangeRate` reads from storage more than necessary
    * [I-4] Constants in `AssetToken` can be set to `private`
    * [I-5] `ThunderLoan::s_feePrecision` should be immutable or constant
    * [I-6] `ThunderLoan__AlreadyAllowed` error can pass the `token` parameter
    * [I-7] `ThunderLoan::setAllowedToken` doesn't verify if the new token has a name and symbol
    * [I-8] `IThunderLoan` interface is imported in `IFlashLoanReceiver` but is not used
    * [I-9] Important functions without `natspec`
    * [I-10] Use of literals instead of constants in `Thunderloan::initialize`

Protocol Summary

The `ThunderLoan` protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

Disclaimer

The LOKAPAL 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
Likelihood	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

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
```

```
4 |   #-- ITSwapPool.sol
5 |   #-- IThunderLoan.sol
6 #-- protocol
7 |   #-- AssetToken.sol
8 |   #-- OracleUpgradeable.sol
9 |   #-- ThunderLoan.sol
10 #-- upgradedProtocol
11     #-- ThunderLoanUpgraded.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

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

This security review was conducted as part of Cyfrin Updraft's Smart Contract Security course.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	8
Info	10
Total	23

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` 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 `assetTokens` and underlying tokens. In a way, it's responsible for keeping track of how many fees to give to liquidity providers.

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

```
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
8  ~~> uint256 calculatedFee = getCalculatedFee(token, amount);
9  ~~> assetToken.updateExchangeRate(calculatedFee);
10
11     token.safeTransferFrom(msg.sender, address(assetToken), amount);
12 }
```

Impact: There are several impacts to this bug:

1. The `redeem` function is blocked, because the protocol thinks the owed tokens is more than it has,
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of Concept:

1. Liquidity provider deposits underlying tokens,
2. User takes out a flash loan,
3. It is now impossible for the liquidity provider to redeem the underlying tokens deposited.

Run this test on your `TunderLoanTest.t.sol` unit suite:

```
1  function testRedeemAfterLoan() public setAllowedToken hasDeposits {
2      uint256 amountToBorrow = AMOUNT * 10;
```

```
3       uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow
            );
4
5       vm.startPrank(user);
6       tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
7       thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
            amountToBorrow, "");
8       vm.stopPrank();
9
10      uint256 amountToRedeem = type(uint256).max;
11
12      vm.startPrank(liquidityProvider);
13      vm.expectRevert();
14      thunderLoan.redeem(tokenA, amountToRedeem);
15      vm.stopPrank();
16  }
```

Recommended mitigation: Consider removing the lines that update the fees in the `deposit` function:

```
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
8   -   uint256 calculatedFee = getCalculatedFee(token, amount);
9   -   assetToken.updateExchangeRate(calculatedFee);
10
11      token.safeTransferFrom(msg.sender, address(assetToken), amount);
12  }
```

[H-2] `ThunderLoan::flashloan` function only checks for returning the money to the protocol, which can be met by calling `deposit` instead of `repay`

Description: The flash loan needs to be repaid in the same transaction. But the `flashloan` function checks for the `endingBalance`. It doesn't have any checks on how the balance get restored. The protocol offers to the user the `repay` function, but any method to sending funds will meet the `endingBalance` requirement, including the `deposit` function:

```
1  if (endingBalance < startingBalance + fee) {
```

```
2       revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
3   }
```

Impact: Any user can create a smart contract that calls the `deposit` function instead of `repay`. This will make them a liquidity provider, allowing them to call `redeem` to drain the protocol's funds.

Proof of Concept: The proof of code for this finding can be found in the `ProofsOfCode.t.sol` file. Copy this file into the project's `./test/unit/` directory and run `forge test --mt testUseDepositInsteadOfRepayToStealFunds -vvvv`.

Recommended mitigation: Consider creating a check on the `flashloan` function that require users to use the proper `repay` method to avoid other ways to restore the `endingBalance`.

[H-3] Mixing up variable location when upgrading contract causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

Description: The `ThunderLoan` contract has two variables in the following order:

```
1   // SLOT 0 is for s_poolFactory from OracleUpgradeable.sol:OracleUpgradeable
2   /*SLOT 1*/ mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
3   /*SLOT 2*/ uint256 private s_feePrecision;
4   /*SLOT 3*/ uint256 private s_flashLoanFee;
```

However, the upgraded contract `ThunderLoanUpgraded.sol` has them in a different order:

```
1   // SLOT 0 is for s_poolFactory from OracleUpgradeable.sol:OracleUpgradeable
2   /*SLOT 1*/ uint256 private s_flashLoanFee;
3   /*SLOT 2*/ uint256 public constant FEE_PRECISION = 1e18;
```

Due to how Solidity works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You **can't** adjust the position of storage variables. And replacing them for constant variables breaks the storage locations as well.

Impact: After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that user who take out flash loans right after an upgrade will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping will point to the wrong storage slot.

Proof of Concept: Run this test on your `TunderLoanTest.t.sol` unit suite:

```
1   // IMPORTS
2   import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
        ThunderLoanUpgraded.sol";
3
4   // TESTS
5   function testUpgradeBreaksStorage() public {
```

```
6       uint256 feeBeforeUpgrade = thunderLoan.getFee();
7
8       vm.startPrank(thunderLoan.owner());
9       ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
10      thunderLoan.upgradeToAndCall(address(upgraded), "");
11      uint256 feeAfterUpgrade = thunderLoan.getFee();
12      vm.stopPrank();
13
14      console.log("feeBeforeUpgrade: ", feeBeforeUpgrade);
15      console.log("feeAfterUpgrade: ", feeAfterUpgrade);
16      assert(feeBeforeUpgrade != feeAfterUpgrade);
17  }
```

Or you can run the same test from the `ProofsOfCode.t.sol` audit file.

Recommended mitigation: Consider leaving a blank storage slot in the upgraded contract to preserve the original order:

```
1  +    uint256 private s_blank;
2       uint256 private s_flashLoanFee;
3       uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using TSwap as a price oracle enables Price and Oracle Manipulation Attacks by reduced fees

Description: The `TSwap` protocol is an automated market maker (AMM) that follows a constant product formula. The price of a token is determined by the reserves on either side of the liquidity pool. Because of this mechanism, malicious users can easily manipulate token prices by executing large buy or sell orders within the same transaction, effectively bypassing protocol fees.

Impact: Liquidity providers will receive significantly lower fees, reducing their incentive to supply liquidity. As a result, new providers may choose to fund other protocols that offer more competitive fee structures.

Proof of Concept: The following steps can proceed in the same transaction:

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`,
2. With that loan they sell 1000 `tokenA` in the `TSwap` protocol, tanking the price,
3. Instead of repaying the loan, the user takes out another flash loan for another 1000 `tokenA`,
4. The fee of the second flash loan will be cheaper, because of how `ThunderLoan` calculates prices based on the `TSwap` protocol.

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

5. The user then repays the first flash loan, and then repays the second flash loan.

NOTICE: The proof of code for this finding can be found in the `ProofsOfCode.t.sol` file. Copy this file into the project's `./test/unit/` directory and run `forge test --mt testOracleManipulation -vvvv`.

Recommended mitigation: Consider using a different price oracle mechanism, like the `Chainlink` price feed with a `Uniswap-TWAP` fallback oracle.

[M-2] `ThunderLoan::initialize` can be called by anyone after deployment, potentially disrupting protocol functionality

Description: The `ThunderLoan::initialize` function is not executed as part of the deployment process, leaving the protocol uninitialized. Additionally, it can be called by anyone at any time, putting the entire protocol at risk.

Impact: The `initialize` function can be front-run by a malicious actor, allowing them to: 1. Set their own `tswapAddress`, 2. Assign themselves as the owner, 3. Reset the `s_flashLoanFee` variable to `3e15` WEI.

Proof of Concept:

1. The protocol is deployed,
2. The `initialize` function is not called by the protocol's owners,
3. A user calls the `initialize` function,
4. The legitimate owners lose proper access control.

Recommended mitigation: Ensure that `initialize` is executed as part of the deployment process to prevent unauthorized access.

Low

[L-1] Reentrancy in `ThunderLoan::flashloan`

Description: There is a potential risk of reentrancy in the `flashloan` function:

```
    // External calls:
    assetToken.updateExchangeRate(fee);
    // Event emitted after the call(s):
    emit FlashLoan(receiverAddress, token, amount, fee, params);
    // State variables written after the call(s):
    s_currentlyFlashLoaning[token] = true;
```

Recommended mitigation: Consider placing the state variable change and the emitted event before the external calls:

```
    // State variables written before the call(s):
+   s_currentlyFlashLoaning[token] = true;
    // Event emitted before the call(s):
+   emit FlashLoan(receiverAddress, token, amount, fee, params);
    // External calls:
    assetToken.updateExchangeRate(fee);
-   emit FlashLoan(receiverAddress, token, amount, fee, params);
-   s_currentlyFlashLoaning[token] = true;
```

[L-2] State variable changes in `ThunderLoan::updateFlashLoanFee` but no event is emitted

Description: State variable changes in the `updateFlashLoanFee` function but no event is emitted:

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

Recommended mitigation: Consider emitting an event for the `s_flashLoanFee` change:

```
// EVENTS
+   event ThunderLoan__updateFlashLoanFee(uint256 newFee);

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

[L-3] `ThunderLoan__ExhangeRateCanOnlyIncrease` custom error not used

Description: The `ThunderLoan__ExhangeRateCanOnlyIncrease` custom error is not used in the contract:

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

Recommended mitigation: It is recommended that the definition be removed when a custom error is unused.

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

[L-4] Centralization risk for trusted owners

Description: The `ThunderLoan` contract has owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds:

```
1 function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns
      (AssetToken)
2
3 function updateFlashLoanFee(uint256 newFee) external onlyOwner
4
5 function _authorizeUpgrade(address newImplementation) internal override
     onlyOwner
```

Recommended mitigation: Consider stating these risks in the documentation, so users are informed of the potential risks.

[L-5] Missing checks for `address(0)` in `OracleUpgradeable::__Oracle_init_unchained`

Description: The `__Oracle_init_unchained` function doesn't check for `address(0)`:

```
1 function __Oracle_init_unchained(address poolFactoryAddress) internal
     onlyInitializing {
2     s_poolFactory = poolFactoryAddress;
3 }
```

Recommended mitigation: Consider checking for `address(0)` when assigning values to address state variables.

```
1 // ERRORS
2 +   error OracleUpgradeable__CantBeZero();
```

```
3
4  // FUNCTIONS
5  function __Oracle_init_unchained(address poolFactoryAddress) internal
       onlyInitializing {
6  +    if (poolFactoryAddress == address(0)) {
7  +        revert OracleUpgradeable__CantBeZero();
8  +    }
9      s_poolFactory = poolFactoryAddress;
10 }
```

[L-6] `public` functions not used internally could be marked `external`

Description: The `ThunderLoan` contract has many functions marked as `public` but not used internally:

```
1      function repay(IERC20 token, uint256 amount) public
2
3      function getAssetFromToken(IERC20 token) public view returns (AssetToken)
4
5      function isCurrentlyFlashLoaning(IERC20 token) public view returns (bool)
```

Recommended mitigation: Instead of marking a function as `public`, consider marking it as `external` if it is not used internally:

```
1  +    function repay(IERC20 token, uint256 amount) external
2
3  +    function getAssetFromToken(IERC20 token) external view returns (AssetToken)
4
5  +    function isCurrentlyFlashLoaning(IERC20 token) external view returns (bool)
6
7  -    function repay(IERC20 token, uint256 amount) public
8
9  -    function getAssetFromToken(IERC20 token) public view returns (AssetToken)
10
11 -    function isCurrentlyFlashLoaning(IERC20 token) public view returns (bool)
```

[L-7] Events with missing `indexed` fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields).

At `AssetToken`:

```
event ExchangeRateUpdated(uint256 newExchangeRate);
```

At `ThunderLoan`:

```
event Deposit(address indexed account, IERC20 indexed token, uint256 amount);

event AllowedTokenSet(IERC20 indexed token, AssetToken indexed asset, bool
    allowed);

event Redeemed(address indexed account, IERC20 indexed token, uint256
    amountOfAssetToken, uint256 amountOfUnderlying);

event FlashLoan(address indexed receiverAddress, IERC20 indexed token, uint256
    amount, uint256 fee, bytes params);
```

Recommended mitigation: Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, every field should be indexed.

[L-8] PUSH0 is not supported by all chains

Description: `Solc` compiler version `0.8.20` switches the default target EVM version to Shanghai, which means that the generated `bytecode` will include PUSH0 opcodes.

Recommended mitigation: Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than `mainnet` like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

Informational

[I-1] The test coverage of the protocol is too low

Description: The coverage of the test suites is too low. It doesn't have any tests for many important functions, for example, `ThunderLoan::redeem`.

Recommended mitigation: Consider adding more test to improve coverage.

[I-2] `ThunderLoan::setAllowedToken` is restricted to the owner, increasing centralization

Description: The `setAllowedToken` function can only be executed by the protocol's owner:

```
function setAllowedToken(IERC20 token, bool allowed)
    external
~~> onlyOwner
    returns (AssetToken)
```

Additionally, this function allows tokens to be removed from the token mapping, which could prevent liquidity providers from redeeming their funds:

```
if (allowed) {
    // HERE IS THE CODE TO ALLOW A NEW TOKEN
} else {
    AssetToken assetToken = s_tokenToAssetToken[token];
~~> delete s_tokenToAssetToken[token];
    emit AllowedTokenSet(token, assetToken, allowed);
    return assetToken;
}
```

Recommended mitigation: Consider restricting the `delete` functionality within the `setAllowedToken` function. If preserving this behavior is necessary, ensure that users are informed about this level of centralization in the documentation.

[I-3] `AssetToken::updateExchangeRate` reads from storage more than necessary

Description: The `updateExchangeRate` function calls the `s_exchangeRate` storage variable five times, increasing gas cost:

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
~~> uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
    totalSupply();

~~> if (newExchangeRate <= s_exchangeRate) {
~~>     revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
    newExchangeRate);
    }
~~> s_exchangeRate = newExchangeRate;
~~> emit ExchangeRateUpdated(s_exchangeRate);
}
```

Recommended mitigation: Consider saving the `s_exchangeRate` value to a temporary variable to avoid multiple storage reads. Also, you can use `newExchangeRate` for the `ExchangeRateUpdated` emit argument:

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
+   uint256 exchangeRate = s_exchangeRate;
```

```
+   uint256 newExchangeRate = exchangeRate * (totalSupply() + fee) /
    totalSupply();
+   if (newExchangeRate <= exchangeRate) {
+       revert AssetToken__ExhangeRateCanOnlyIncrease(exchangeRate,
    newExchangeRate);
+   }
+   s_exchangeRate = newExchangeRate;
+   emit ExchangeRateUpdated(newExchangeRate);

-   uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
    totalSupply();
-   if (newExchangeRate <= s_exchangeRate) {
-       revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
    newExchangeRate);
-   }
-   s_exchangeRate = newExchangeRate;
-   emit ExchangeRateUpdated(s_exchangeRate);
}
```

[I-4] Constants in `AssetToken` can be set to `private`

Description: In the `AssetToken` contract, the `EXCHANGE_RATE_PRECISION` variable is set to `public`:

```
    uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

Recommended mitigation: Consider setting the constant to `private` and adding a getter function:

```
// CONSTANTS
+   uint256 private constant EXCHANGE_RATE_PRECISION = 1e18;
-   uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;

// GETTER FUNCTIONS
+   function getExchangeRatePrecision() external pure returns (uint256) {
+       return EXCHANGE_RATE_PRECISION;
+   }
```

[I-5] `ThunderLoan::s_feePrecision` should be immutable or constant

Description: The `s_feePrecision` variable uses a storage slot, but is not updated after `initialize`:

```
    uint256 private s_feePrecision;
```

Recommended mitigation: Consider changing it to constant or immutable:

```
+   uint256 private constant FEE_PRECISION = 1e18;
-   uint256 private s_feePrecision;
```

[I-6] `ThunderLoan__AlreadyAllowed` error can pass the `token` parameter

Description: The `setAllowedToken` function uses the `ThunderLoan__AlreadyAllowed` custom error. This function takes a token as a parameter to be evaluated:

```
if (address(s_tokenToAssetToken[token]) != address(0)) {
    revert ThunderLoan__AlreadyAllowed();
}
```

It would be beneficial if the evaluated token was passed when the function reverts with the custom error.

Recommended mitigation: Consider adding the `token` parameter in the custom error:

```
// ERRORS
+   error ThunderLoan__AlreadyAllowed(IERC20 token);
-   error ThunderLoan__AlreadyAllowed();

// setAllowedtoken FUNCTION
if (address(s_tokenToAssetToken[token]) != address(0)) {
+   revert ThunderLoan__AlreadyAllowed(token);
-   revert ThunderLoan__AlreadyAllowed();
}
```

[I-7] `ThunderLoan::setAllowedToken` doesn't verify if the new token has a name and symbol

Description: The `setAllowedToken` function creates a new `assetToken` using a name and symbol derived from a combination of the `ThunderLoan` / `tl` prefix and the underlying token's name and symbol:

```
string memory name = string.concat("ThunderLoan ", IERC20Metadata(address(token
    )).name());
string memory symbol = string.concat("tl", IERC20Metadata(address(token)).
    symbol());
AssetToken assetToken = new AssetToken(address(this), token, name, symbol);
```

The protocol is designed to support `USDC`, `DAI`, `LINK` and `WETH` at launch. However, if an upgrade introduces tokens that lack a name and/or symbol, the `string.concat` function may fail to generate

proper metadata for the new `assetToken`.

Recommended mitigation: Consider adding checks in `setAllowedToken` to ensure that newly added tokens have valid metadata, or restrict allowed tokens to those with properly defined name and symbol fields.

[I-8] `IThunderLoan` interface is imported in `IFlashLoanReceiver` but is not used

Description: The `IFlashLoanReceiver` contract imports the `IThunderLoan` interface but is not used. The only place in which `IFlashLoanReceiver` is imported and the `repay` function is called is in the `MockFlashLoanReceiver` contract. And even there the `repay` function is used directly with the `IThunderLoan` interface.

Recommended mitigation: Consider removing the `IThunderLoan` interface import after checking if it was intended to be implemented in another way.

[I-9] Important functions without `natspec`

Description: There are multiple important functions that don't have a `natspec` in multiple contracts:

- `AssetToken::constructor`
- `ThunderLoan::initialize`
- `ThunderLoan::deposit`
- `ThunderLoan::flashloan`
- `ThunderLoan::repay`
- `ThunderLoan::setAllowedToken`

Recommended mitigation: Consider adding the `natspec` in those functions.

[I-10] Use of literals instead of constants in `Thunderloan::initialize`

Description: The `initialize` function uses literal numbers (or "magic numbers") instead of constants for the `s_feePrecision` and `s_flashLoanFee` arguments:

```
1 function initialize(address tswapAddress) external initializer {
2     __Ownable_init(msg.sender);
3     __UUPSUpgradeable_init();
4     __Oracle_init(tswapAddress);
5 ~~> s_feePrecision = 1e18;
6 ~~> s_flashLoanFee = 3e15;
7 }
```

Recommended mitigation: Consider replacing literals for constants:

```
// STATE VARIABLES
+   uint256 private constant PRECISION = 1e18;
+   uint256 private constant FEE = 3e15;

// FUNCTIONS
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
+   s_feePrecision = PRECISION;
+   s_flashLoanFee = FEE;
-   s_feePrecision = 1e18;
-   s_flashLoanFee = 3e15;
}
```