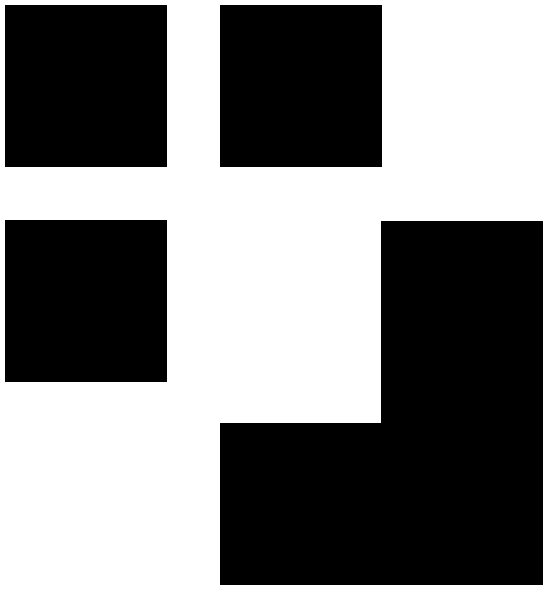

Protocol Audit Report

Version 1.0

Hexific.com

November 8, 2025

Protocol Audit Report

Hexific.com

November 8, 2025

Prepared by: Hexific Lead Security Researcher: - Febri Nirwana

Table of Contents

Protocol Summary

The ThunderLoan protocol is a decentralized flash loan platform built on Ethereum that allows users to borrow assets without collateral, provided they return the borrowed amount plus a fee within a single transaction. The protocol operates through an upgradeable proxy pattern (UUPS) and integrates with TSwap for price oracle functionality.

Disclaimer

The Hexific 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 the following commit hash:

```
2250d81b89aebdd9cb135382e068af8c269e3a4b
```

Scope

```
./interfaces/
#-- IFlashLoanReceiver.sol
#-- IPoolFactory.sol
#-- ITSwapPool.sol
#-- IThunderLoan.sol

./protocol/
#-- AssetToken.sol
#-- OracleUpgradeable.sol
```

```
10 #-- ThunderLoan.sol
11
12 ./upgradedProtocol/
13 #-- 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 Hexific team conducted a security audit of the ThunderLoan protocol smart contracts to evaluate their design, implementation, and resistance to common attack vectors. Our review identified several critical issues that could lead to significant financial losses for liquidity providers and protocol insolvency.

The most severe findings include an erroneous exchange rate update in the deposit function that causes protocol accounting corruption and blocks redemptions, and a storage collision vulnerability in the upgrade mechanism that corrupts critical state variables. Additionally, we found centralization risks with blacklistable tokens, initialization front-running vulnerabilities, missing event emissions, and various code quality issues.

The protocol implements a flash loan lending platform with upgradeable proxy architecture but contains fundamental flaws in its fee accounting and upgrade safety that make it unsafe for production use in its current state. Immediate remediation of the High severity findings is strongly recommended before any deployment.

Issues found

Severity	Number of Issues Found
High	2
Medium	1
Low	4
Gas	2

Severity	Number of Issues Found
Info	11
Total	20

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 redemptions 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!

```
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      // @Audit-High
9  @>  uint256 calculatedFee = getCalculatedFee(token, amount);
10 @>  assetToken.updateExchangeRate(calculatedFee);
11
12     token.safeTransferFrom(msg.sender, address(assetToken), amount);
13 }
```

Impact:

- 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:

1. LP deposits
2. User takes out a flash loan
3. It is now impossible for LP to redeem

PoC Code

Place the following into `ThunderLoanTest.t.sol`:

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

    vm.startPrank(user);
    tokenA.mint(address(mockFlashLoanReceiver), AMOUNT); // we should
        mint here to cover the fee
    thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
        amountToBorrow, "");
    vm.stopPrank();

    vm.prank(liquidityProvider);
    thunderLoan.redeem(tokenA, DEPOSIT_AMOUNT);

    uint256 expectedBalance = DEPOSIT_AMOUNT + calculatedFee;
    assertEq(tokenA.balanceOf(liquidityProvider), expectedBalance);
}
```

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-2] Storage collision due to incorrect variable ordering in ThunderLoan upgrade causes protocol to freeze and corrupts critical state

Description:

`ThunderLoan.sol` has two variables in the following order:

```
1     uint256 private s_feePrecision;
2     uint256 private s_flashLoanFee; // 0.3% ETH fee
```

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

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

The upgraded contract reorders state declarations (and inserts a constant) relative to the deployed ThunderLoan, which shifts storage slots under the proxy because constants do not occupy storage. As a result, previously stored values are read into the wrong variables after upgrade.

Impact:

Fees and mapping `s_currentlyFlashLoaning` become corrupted, breaking fee calculations and flashloan control flow and causing reverts or denial-of-service for core operations. This makes upgrades high-risk and can lock or misaccount user funds.

Proof of Concept:

1. Check fee before upgrade
2. Upgrade proxy
3. Check expected fee slot after upgrade

PoC Code

Place the following into `ThunderLoanTest.t.sol`:

```
1  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
2  .
3  .
4  .
5  function testUpgradeBreaksFee() public setAllowedToken hasDeposits {
6      uint256 feeBeforeUpgrade = thunderLoan.getFee();
7      vm.startPrank(thunderLoan.owner());
8      ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
9      thunderLoan.upgradeToAndCall(address(upgraded), "");
10     uint256 feeAfterUpgrade = thunderLoan.getFee();
11     vm.stopPrank();
```

```
12
13      console2.log("Fee before upgrade:", feeBeforeUpgrade);
14      console2.log("Fee after upgrade:", feeAfterUpgrade);
15      assert(feeBeforeUpgrade != feeAfterUpgrade);
16  }
```

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

Recommended Mitigation:

If you must remove the storage variable, leave it as blank as to not mess up the storage slots.

```
1  -    uint256 private s_flashLoanFee; // 0.3% ETH fee
2  -    uint256 public constant FEE_PRECISION = 1e18;
3  +    uint256 private s_blank;
4  +    uint256 private s_flashLoanFee;
5  +    uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Centralized tokens with blacklist functionality (e.g., USDC) can freeze `ThunderLoan` and `AssetToken` contracts, causing protocol denial-of-service

Description:

Many popular stablecoins like USDC implement address blacklisting, allowing their issuers to block transfers to/from specific addresses. If the issuer blacklists either the `ThunderLoan` contract or any `AssetToken` contract, all operations involving that token will revert. The protocol calls `safeTransfer` and `safeTransferFrom` in critical paths (deposit, redeem, flashloan repayment, and `AssetToken::transferUnderlyingTo`), so a blacklist would brick those flows entirely.

Relevant code in `AssetToken.sol`:

```
1  function transferUnderlyingTo(address to, uint256 amount) external
       onlyThunderLoan {
2      // @audit-medium the protocol will be frozen if USDC blacklists
           this contract
3  @>  i_underlying.safeTransfer(to, amount);
4  }
```

And in `ThunderLoan.sol`:

```
1  function deposit(IERC20 token, uint256 amount) external ... {
2      ...
3  @>  token.safeTransferFrom(msg.sender, address(assetToken), amount); //
       reverts if AssetToken is blacklisted
```

```
4 }
```

Impact:

Blacklisting causes a full denial-of-service for the affected token: deposits, redemptions, and flash-loan repayments revert, trapping user funds in the AssetToken with no on-chain withdrawal path. This halts liquidity and fee generation, erodes user confidence, and may lead to permanent loss of liquidity unless the blacklist is lifted or a recovery mechanism exists.

Recommended Mitigation:

Consider documenting this risk in protocol materials and implementing an emergency withdrawal mechanism (owner-gated or governance-controlled) that transfers underlying tokens to a recovery address if blacklisted, allowing users to claim funds off-chain. Add monitoring and alerts for blacklist events to enable quick response through market pausing or coordination with token issuers.

Low

[L-1] The Front Running attack potential from `ThunderLoan::initialize`, when this be call later after contract deployment

Description:

The `ThunderLoan::initialize` function is used to set up critical protocol parameters including the owner and the TSwap pool factory address. Because this function uses the `initializer` modifier from OpenZeppelin's upgradeable contracts, it can only be called once.

However, if the contract is deployed and the legitimate owner does not call `initialize` immediately, an attacker can front-run the transaction and call `initialize` first with their own address and a malicious pool factory.

Impact:

If an attacker successfully front-runs the initialization, they become the owner of the `ThunderLoan` protocol and can control all privileged functions. This allows them to drain funds by setting malicious asset tokens, manipulating the oracle price feed through a fake pool factory, or upgrading the contract to a malicious implementation.

Proof of Concept:

PoC Code

First, comment this line code in `BaseTest.t.sol`:

```
1 -   thunderLoan.initialize(address(mockPoolFactory));
2 +   // thunderLoan.initialize(address(mockPoolFactory));
```

Then, place the following into `ThunderLoanTest.t.sol`:

```
function testFrontRunning() public {
    address attacker = makeAddr("attacker");
    MockPoolFactory attackerMockPoolFactory = new MockPoolFactory();

    vm.prank(attacker);
    thunderLoan.initialize(address(attackerMockPoolFactory));
    assert(attacker == thunderLoan.owner());
}
```

Recommended Mitigation:

Call `initialize` in the same transaction as contract deployment (using a deployment script or factory pattern), or implement a two-step initialization where the deployer address is hardcoded in the constructor and only that address can call `initialize`.

```
+ address private immutable i_deployer;
+
  constructor() {
+     i_deployer = msg.sender;
      _disableInitializers();
  }

  function initialize(address tswapAddress) external initializer {
+     require(msg.sender == i_deployer, "Only deployer can initialize")
    ;
      __Ownable_init(msg.sender);
      __UUPSUpgradeable_init();
      __Oracle_init(tswapAddress);
      s_feePrecision = 1e18;
      s_flashLoanFee = 3e15;
  }
```

[L-2] Both `ThunderLoan::updateFlashLoanFee` and `ThunderLoanUpgraded::updateFlashLoanFee` Missing Emit Event

Description:

The `updateFlashLoanFee` function in both contracts does not emit an event when the flash loan fee is updated. This can lead to a lack of transparency and make it difficult for external observers to track changes to important state variables.

Impact:

Without an event to signal when the flash loan fee is updated, it becomes harder for users and other contracts to react to changes in the fee structure. This could lead to unexpected behavior or exploitation by malicious actors who are able to front-run transactions.

Recommended Mitigation:

Add an event declaration and emit the event in the `updateFlashLoanFee` function:

```
+ event FlashLoanFeeUpdated(uint256 oldFee, uint256 newFee);
.
.
.
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
    if (newFee > s_feePrecision) {
        revert ThunderLoan__BadNewFee();
    }
    s_flashLoanFee = newFee;
+   emit FlashLoanFeeUpdated(oldFee, newFee);
}
```

[L-3] `ThunderLoan` and `ThunderLoanUpgraded` Centralization Risk

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 244

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

- Found in src/protocol/ThunderLoan.sol Line: 270

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
        {
```

- Found in src/protocol/ThunderLoan.sol Line: 298

```
    function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 238

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 264

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
        {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

```
    function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

[L-4] `IFlashLoanReceiver` Unused Import

Redundant import statement. Consider removing it.

1 Found Instances

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 4

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

Gas

[G-1] `ThunderLoan` states should be immutable

Description:

The state variables `s_feePrecision` and `s_flashLoanFee` in `ThunderLoan.sol` are only set once in the `initialize` function and never modified afterward (except through the `updateFlashLoanFee` function). Consider making `s_feePrecision` immutable or constant to save gas and improve code clarity.

Recommended Mitigation:

Convert `s_feePrecision` to a constant since it's set to a fixed value. For `s_flashLoanFee`, if updates are needed, keep it as-is; otherwise, document why it remains mutable.

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

[G-2] `OracleUpgradeable::getPrice` redundant function

Description:

The `getPrice` function is a simple wrapper that calls `getPriceInWeth` with no additional logic, adding unnecessary bytecode and gas overhead. This increases contract size and deployment costs without providing value.

Recommended Mitigation:

Remove the `getPrice` function and use `getPriceInWeth` directly:

```
- function getPrice(address token) external view returns (uint256) {
-     return getPriceInWeth(token);
- }
```

Information

[I-1] Missmatch `ThunderLoan::initialize` parameter name

Description:

The `initialize` function parameter is named `tswapAddress` but it represents a pool factory address, not a TSwap pool address. This naming inconsistency can confuse developers and auditors.

Recommended Mitigation:

Rename the parameter to accurately reflect its purpose:

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

[I-2] `ThunderLoan::initialize` magic number

Description:

The `initialize` function uses magic numbers `1e18` and `3e15` without explanation or named constants, making the code harder to understand and maintain.

Recommended Mitigation:

Define named constants at the contract level:

```
+ uint256 private constant STARTING_FEE_PRECISION = 1e18;
+ uint256 private constant STARTING_FLASH_LOAN_FEE = 3e15; // 0.3%

  function initialize(address poolFactoryAddress) external initializer
      {
        __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();
```

```
7         __Oracle_init(poolFactoryAddress);
8 -       s_feePrecision = 1e18;
9 -       s_flashLoanFee = 3e15;
10 +      s_feePrecision = STARTING_FEE_PRECISION;
11 +      s_flashLoanFee = STARTING_FLASH_LOAN_FEE;
12    }
```

[I-3] `ThunderLoan` missing natspec

Description:

Critical functions `deposit`, `flashloan`, and `setAllowedToken` lack NatSpec documentation, making it harder for developers and auditors to understand their purpose, parameters, and return values. Also increases onboarding time for new developers, and can lead to misuse of the contract's functions.

Recommended Mitigation:

Add comprehensive NatSpec comments to all public and external functions:

```
1 /// @notice Allows users to deposit tokens and receive asset tokens in
      return
2 /// @param token The ERC20 token to deposit
3 /// @param amount The amount of tokens to deposit
4 function deposit(IERC20 token, uint256 amount) external { ... }
```

[I-4] `ThunderLoan::flashloan` messed up with slither disables

Description:

The `flashloan` function contains multiple `slither-disable-next-line` comments that disable important static analysis checks without clear justification, potentially hiding security issues. This can mask legitimate security concerns and make future audits more difficult.

Recommended Mitigation:

Either address the underlying issues flagged by Slither or add detailed comments explaining why each specific warning is safe to ignore in this context.

[I-5] `ThunderLoan::ThunderLoan__AlreadyAllowed` should revert with token address

Description:

The `ThunderLoan__AlreadyAllowed` error is thrown when trying to add a token that's already allowed, but it doesn't include the token address in the error, making debugging harder.

Recommended Mitigation:

Update the error to include the token address:

```
- error ThunderLoan__AlreadyAllowed();
+ error ThunderLoan__AlreadyAllowed(address token);

  function setAllowedToken(IERC20 token, bool allowed) external
      onlyOwner {
      if (allowed) {
          if (address(s_tokenToAssetToken[token]) != address(0)) {
-             revert ThunderLoan__AlreadyAllowed();
+             revert ThunderLoan__AlreadyAllowed(address(token));
          }
      }
  }
```

[I-6] Unused Error `ThunderLoan` and `ThunderLoanUpgraded`

Consider using or removing the unused error.

2 Found Instances

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

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

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

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[I-7] `OracleUpgradeable::__Oracle_init_unchained` need to do zero address check

Description:

The initialization function `__Oracle_init_unchained` accepts a `poolFactoryAddress` parameter but doesn't validate that it's not the zero address, would cause all price lookups to fail, breaking core protocol functionality with no way to fix it without redeployment.

Recommended Mitigation:

Add a zero address check:

```
1    function __Oracle_init_unchained(address poolFactoryAddress) internal
         onlyInitializing {
2  +     require(poolFactoryAddress != address(0), "OracleUpgradeable:
       zero address");
3        s_poolFactory = poolFactoryAddress;
4    }
```

[I-8] `OracleUpgradeable::getPriceInWeth` should try to implement forked test

Description:

The `getPriceInWeth` function calls external contracts (TSwap pools) but lacks integration tests using forked mainnet/testnet data to verify behavior against real pool contracts.

Recommended Mitigation:

Implement forked tests to validate price fetching logic against actual TSwap deployments and catch integration issues early.

[I-9] `IThunderLoan` should be implement by the `ThunderLoan` contract

Description:

The `ThunderLoan` contract doesn't explicitly implement the `IThunderLoan` interface, making it harder to verify interface compliance and increasing the risk of signature mismatches.

Recommended Mitigation:

Explicitly implement the interface:

```
1  - contract ThunderLoan is Initializable, OwnableUpgradeable,
       UUPSUpgradeable, OracleUpgradeable {
2  + contract ThunderLoan is Initializable, OwnableUpgradeable,
       UUPSUpgradeable, OracleUpgradeable, IThunderLoan {
```

[I-10] `IThunderLoan::repay` different parameter types

Description:

The `repay` function signature in `IThunderLoan` may have different parameter types than the actual implementation, leading to ABI compatibility issues, failed external calls and integration problems.

Recommended Mitigation:

Ensure the interface exactly matches the implementation:

```
- function repay( token, uint256 amount) public {
+ function repay(address token, uint256 amount) public {
-     if (!s_currentlyFlashLoaning[token]) {
+     if (!s_currentlyFlashLoaning[IERC20(token)]) {
          revert ThunderLoan__NotCurrentlyFlashLoaning();
      }
-     AssetToken assetToken = s_tokenToAssetToken[token];
+     AssetToken assetToken = s_tokenToAssetToken[IERC20(token)];
-     token.safeTransferFrom(msg.sender, address(assetToken), amount);
+     IERC20(token).safeTransferFrom(msg.sender, address(assetToken),
    amount);
  }
```

[I-11] `IFlashLoanReceiver` missing natspec

Description:

The `IFlashLoanReceiver` interface lacks NatSpec documentation explaining the purpose of `executeOperation` and its parameters. This may lead to misunderstandings of requirements, resulting in incorrect flash loan receiver implementations.

Recommended Mitigation:

Add comprehensive NatSpec to the interface:

```
/// @notice Called by ThunderLoan after transferring flash loan funds
/// @param token The token that was borrowed
/// @param amount The amount borrowed
/// @param fee The fee to be paid
/// @param initiator The address that initiated the flash loan
/// @param params Arbitrary data passed from the initiator
/// @return Must return true for the flash loan to succeed
function executeOperation(...) external returns (bool);
```