Smart Contract Audit

Date : 2024-11-14

EmpowerCoin Audit

Executive Summary

Type	Smart Contract Audit
Audit Timeline	14 days
Runtime Environment	EVM
Languages	Solidity

Scope

./Swap/*

./UpgradeableContracts/*

Summary of Findings

ID	Name	Description	Severity
H-01	Position Not Closed After function call in EmpowerToken::withdrawRewardsWhileRemovingPosition() function	The withdrawRewardsWhileRemovingPosition function fails to close the position after processing the rewards withdrawal.	High
H-02	Centralized Ownership and Privilege Risks	All contracts use OpenZeppelin's Ownable for access control, which centralizes critical privileges to a single owner.	High
H-03	Incorrect Balance Update in Case of Failed Transfers in withdrawOwnRewards	The function withdrawOwnRewards updates the user's payout balances before attempting transfers of rewards	High
H-04	Missing Storage Gaps in UUPS Contracts	Storage gaps are essential to ensure safe upgrades by preventing storage collisions when new state variables are added in the future.	High
M-01	Missing Sanity Checks for Stale or Incorrect Data	The contracts rely on Oracle data fetched using methods like latestRoundData() from Chainlink or other oracles but do not verify the freshness of the data.	Medium

M–02	Missing Use of safeTransfer and safeTransferFrom for ERC20 Transfers	Using transfer and transferFrom directly can lead to vulnerabilities if the token being transferred does not conform to the ERC20 standard.	Medium

Findings

[H-01]Position Not Closed After function call in EmpowerToken::withdrawRewardsWhileRemovingPosition() function

Severity: High

Location: EmpowerTokenUpgradeable.sol

Description:

The `withdrawRewardsWhileRemovingPosition` function fails to close the position after processing the rewards withdrawal. This oversight allows the function to be called repeatedly for the same position, potentially draining the contract's funds. Since the function does not update the position state to prevent further calls, malicious users can exploit this vulnerability.

Impact:

- **Repeated Calls:** Attackers can repeatedly call the function, claiming rewards multiple times for the same position.
- **Funds Depletion:** The protocol's funds could be completely drained, leading to financial loss for the protocol and users.
- **Protocol Disruption:** Legitimate users might face delays or loss of rewards as the contract's resources are exhausted.

Recommendation:

1. Update the position state to mark it as "closed" or "claimed" after processing the rewards.
2. Add a validation check to ensure that the position is not already claimed before processing the rewards.

Proof of Concept

```
function withdrawRewardsWhileRemovingPosition(uint256 positionId) external
returns (bool) {
       (bool success, bytes memory data) =
s_borrowLend.staticcall(abi.encodeWithSignature("getPosition(uint256)",
positionId));

       IBorrowLend.Position memory position = abi.decode(data,
(IBorrowLend.Position));

       if (!position.removing) {
           revert EmpowerToken__PositionNotRemoving();
       }

       // get the Reward
       uint256 empowerRewards =
empowerDividendsOfBorrower(position.borrower, positionId);
       uint256 ethRewards = dividendsOfBorrower(position.borrower,
positionId);
       uint256 usdcRewards = usdcDividendsOfBorrower(position.borrower,
positionId);

       if (empowerRewards + ethRewards + usdcRewards == 0) {
           return true;
       }

       payoutsTo_[position.borrower] += (int256)(ethRewards * magnitude);
       payoutsToUSDC_[position.borrower] += (int256)(usdcRewards *
magnitude);
       payoutsToEmpower_[position.borrower] += (int256)(empowerRewards *
magnitude);

       (success,) = payable(address(position.borrower)).call{value:
ethRewards}("");

       if (!success) {
           revert EmpowerToken__TransferFailed();
       }

       try IERC20(i_usdcToken).transfer(position.borrower, usdcRewards) {}
catch {}
```

```
        transfer_(address(this), position.borrower, empowerRewards); //
@audit consider using safeTransfer

        // emit the event.
        emit rewardClaimed(
            position.borrower,
            ethRewards,
            usdcRewards,
            empowerRewards,
            tokenBalanceLedger_[position.borrower],
            payoutsTo_[position.borrower],
            payoutsToUSDC_[position.borrower],
            payoutsToEmpower_[position.borrower]
        );

        return true;
    }
```

[H-02] Centralized Ownership and Privilege Risks

Severity: High

Location: All contracts

Description:

All contracts use OpenZeppelin's `Ownable` for access control, which centralizes critical privileges to a single owner. This poses a significant risk if the owner key is compromised, lost, or abused.

Impact:

1) **Single Point of Failure:** The protocol's functionality and funds could be compromised if the owner's private key is exposed.
2) **Centralization Concerns:** This conflicts with the decentralized ethos of blockchain systems.

Recommendation

Replace `Ownable` with a multi-sig wallet for key administrative functions.
Consider implementing a DAO-based governance model for decentralized decision-making.
Add a time delay for critical functions, allowing community members to react to suspicious actions.

[H-03] Incorrect Balance Update in Case of Failed Transfers in `withdrawOwnRewards`

Severity: High

Location: EmpowerTokenUpgradeable.sol

Description:

The function `withdrawOwnRewards` updates the user's payout balances before attempting transfers of rewards (ETH, USDC, and Empower tokens). If any transfer fails, the user's balances are already updated, causing an inconsistency between the user's actual rewards and the recorded payout balances. This can lead to permanent loss of funds for the user as the rewards are marked as "claimed" even though the user hasn't received them.

Impact:

1) **Permanent Fund Loss:** Users will lose their rewards due to the premature update of `payoutsTo_`, `payoutsToUSDC_`, and `payoutsToEmpower_`.

2) **Data Inconsistency:** The recorded balances will not accurately reflect the actual funds transferred, leading to discrepancies in the reward tracking mechanism.

3) **Loss of Trust:** Users may lose confidence in the protocol due to incorrect handling of funds.

Recommendation

1) **Defer Balance Updates:** Update `payoutsTo_`, `payoutsToUSDC_`, and `payoutsToEmpower_` only after successfully transferring the rewards.

```
// Perform transfers first
(bool success,) = payable(address(user)).call{value: ethRewards}("");
require(success, "ETH transfer failed");

try IERC20(i_usdcToken).transfer(user, usdcRewards) {
   // success
} catch {
   revert("USDC transfer failed");
}

transfer_(address(this), user, empowerRewards);

// Only update balances after all transfers succeed
payoutsTo_[user] += (int256)(ethRewards * magnitude);
payoutsToUSDC_[user] += (int256)(usdcRewards * magnitude * (10 **
additionalPrecision));
payoutsToEmpower_[user] += (int256)(empowerRewards * magnitude);
```

2) **Handle Failures Gracefully:** If any transfer fails, revert the transaction to ensure that the user's balances are not updated incorrectly.

3) **Audit Redundant Checks:** Remove the unnecessary `success` check after `transfer_` since `require` ensures all conditions are met before proceeding.

Proof Of Concept :

```
function withdrawOwnRewards() external {
        address user = msg.sender;
        uint256 ethRewards = dividendsOf(user);
        uint256 usdcRewards = usdcDividendsOf(user);
        uint256 empowerRewards = empowerDividendsOf(user);

        uint256 additionalPrecision = decimals() -
ERC20Upgradeable(i_usdcToken).decimals();
```

```
        if (ethRewards + (usdcRewards * (10 ** additionalPrecision)) +
empowerRewards <= 0) {
            revert EmpowerToken__RewardsNotFound();
        }

        // update the payouts
        // update dividend tracker
        payoutsTo_[user] += (int256)(ethRewards * magnitude);
        payoutsToUSDC_[user] += (int256)(usdcRewards * magnitude * (10 **
additionalPrecision));
        payoutsToEmpower_[user] += (int256)(empowerRewards * magnitude);

        // transfer the amount
        (bool success,) = payable(address(user)).call{value:
ethRewards}("");

        if (!success) {
            revert EmpowerToken__TransferFailed();
        }

        try IERC20(i_usdcToken).transfer(user, usdcRewards) {} catch {} //
@audit Med this will make failing transactions due to any reason

        transfer_(address(this), user, empowerRewards);

        if (!success) {
            revert EmpowerToken__TransferFailed();
        } // @audit no need for this check as its already been done

        // emit the event.
        emit rewardClaimed(
            user,
            ethRewards,
            usdcRewards,
            empowerRewards,
            tokenBalanceLedger_[user],
            payoutsTo_[user],
            payoutsToUSDC_[user],
            payoutsToEmpower_[user]
        );
    }
```

[H-04] Missing Storage Gaps in UUPS Contracts

Severity: High

Location: All contracts

Description:

The UUPS upgradeable contracts `LoanPayment`, `EmpowerToken`, and `Treasury` do not include storage gaps in their implementation. Storage gaps are essential to ensure safe upgrades by preventing storage collisions when new state variables are added in the future. Without storage gaps, upgrades can inadvertently overwrite existing variables, leading to storage corruption and unexpected behaviors.

Impact:

1) **Upgradability Risk:** Future upgrades introducing new state variables may overwrite existing storage, leading to unintended consequences.
2) **Data Corruption:** Stored data could be lost or corrupted during an upgrade, disrupting protocol functionality.
3) **Increased Development Complexity:** Without gaps, developers need to carefully plan storage layouts for upgrades, increasing the potential for errors.

Recommendation

To address this, a robust solution is to create a **base contract** with storage gaps that can be inherited by all UUPS upgradeable contracts. This approach centralizes the storage gap management and simplifies maintenance across the protocol.

Proposed Fix:

1) Create a base contract

```
pragma solidity ^0.8.0;

contract BaseUpgradeable {
   // Reserved storage slots for future upgrades
   uint256[50] private __gap;
```

2) Inherit this base contract in all upgradeable contracts:

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

import "./BaseUpgradeable.sol";

contract LoanPayment is BaseUpgradeable, Initializable,
   OwnableUpgradeable, ReentrancyGuardUpgradeable, UUPSUpgradeable {
   // Contract implementation
}
```

[M-01] Missing Sanity Checks for Stale or Incorrect Data

Severity: Medium

Location: BorrowLend.sol

Description:

The contracts rely on Oracle data fetched using methods like **latestRoundData**() from Chainlink or other oracles but do not verify the freshness of the data. Oracle feeds can return stale or outdated prices due to various reasons such as oracle downtimes, chain disruptions, or delayed updates. Without proper sanity checks for staleness, these contracts are exposed to using invalid or outdated price data for critical calculations, leading to inaccurate transactions or potential loss of funds.

Impact:

Stale Pricing Data Usage:

- Operations relying on outdated prices could result in overpayment or underpayment, impacting lenders, borrowers, and liquidity providers.
- Borrowers may repay loans with incorrect amounts, either benefiting themselves unfairly or overpaying and causing losses.

Systemic Risks:

- Malicious actors could exploit stale or incorrect data to manipulate the protocol, drain funds, or execute unintended operations.

Degraded Trust:

- Users might lose trust in the system due to inaccurate outcomes from transactions involving stale or incorrect price data.

Recommendation

1. **Implement Staleness Checks:**
   - Use the `updatedAt` parameter returned by `latestRoundData()` and compare it to a predefined staleness threshold (e.g., `block.timestamp - heartbeat`).
     `(, int256 price, , uint256 updatedAt, ) = priceFeed.latestRoundData();`

```
if (updatedAt < block.timestamp - 1 hours) {
    revert("Stale oracle price data");
}
```

   - Use appropriate thresholds specific to each feed by checking their heartbeat intervals.
2. **Check for Oracle Reverts:**

    - Wrap oracle calls in `try/catch` to handle potential reverts gracefully and use fallback mechanisms for critical operations.

3. **Handle Incorrect Values:**
    - Validate the range of returned prices using `minAnswer` and `maxAnswer` from Chainlink feeds (or similar parameters for other oracles).

```
require(price > minPrice && price < maxPrice, "Oracle price out
of bounds");
```

4. **Different Heartbeats for Different Feeds:**
    - Ensure each price feed's staleness threshold matches its specific heartbeat interval. Avoid using a single global threshold for multiple feeds.
5. **Off-chain Monitoring:**
    - Use off-chain systems to verify Oracle prices against other sources, and flag or disable feeds reporting unexpected values.

Proof of Concept :

```
function convertEthToUsd() public view returns (uint256) {
       (
           /* uint80 roundID */
           ,
           int256 answer,
           /*uint startedAt*/
           ,
           /*uint timeStamp*/
           ,
           /*uint80 answeredInRound*/
       ) = s_ethUsdPriceFeed.latestRoundData();

       uint256 additionalPrecision = s_ethUsdPriceFeed.decimals() -
ERC20Upgradeable(s_USDCTokenAddress).decimals();
       return uint256(answer) / (10 ** additionalPrecision);
   }
```

[M-02] Missing Use of `safeTransfer` and `safeTransferFrom` for ERC20 Transfers

Severity: Medium

Location: All Contracts

Description:

The contract uses low-level `transfer` and `transferFrom` functions for transferring ERC20 tokens. For example:

```
bool transferSuccess =
IERC20(_lendingTokenAddress).transferFrom(msg.sender,
address(this), totalAmountToPay);
```

Using `transfer` and `transferFrom` directly can lead to vulnerabilities if the token being transferred does not conform to the ERC20 standard. Specifically:

- Some tokens do not return a boolean value, and instead revert on failure. In such cases, the low-level call will silently succeed, leading to unintended outcomes.
- Malicious tokens could exploit this behavior to bypass transfer checks, potentially allowing unauthorized token transfers.

Impact:

- **Funds Loss:** Tokens that do not adhere to the ERC20 standard may fail silently, leading to discrepancies in token balances.
- **Denial of Service:** Non-compliant tokens can cause unexpected behavior in the contract, leading to a potential denial of service.
- **Security Risks:** Malicious tokens can exploit this vulnerability to mislead the contract into believing a transfer succeeded when it did not.

Proof of Concept (PoC):

Consider the following scenario:

1. A non-compliant token does not revert on failure but instead returns a non-boolean value or no value at all.

2. The contract, using the low-level `transfer` or `transferFrom`, assumes the transfer succeeded.
3. This could lead to incorrect updates to the contract's internal accounting, resulting in discrepancies or loss of funds.

Recommendation:

Use the `safeTransfer` and `safeTransferFrom` functions provided by OpenZeppelin's `SafeERC20` library. These functions ensure proper handling of token transfers and throw meaningful errors if the token does not conform to the ERC20 standard.

Proof of Concept:

```
IERC20(_lendingTokenAddress).safeTransferFrom(msg.sender, address(this),
totalAmountToPay);
```

[L-01] Redundant Check for Transfer Success in `withdrawOwnRewards`

Severity: Low

Location: EmpowerTokenUpgradeable.sol

Description: In the `withdrawOwnRewards` function, the following check is redundant as it repeats a validation that has already been performed earlier in the function:

if (!success) { revert EmpowerToken__TransferFailed();

The `success` variable is reassigned after the ETH transfer and is not used for any other purpose beyond this point. However, the logic flow ensures that transfer failure would already result in a revert earlier in the function, making this additional check unnecessary.

Impact:

1) **Gas Wastage:** Redundant checks unnecessarily increase the gas cost of the function.

2) **Code Clutter:** Maintaining redundant checks can make the code harder to read and maintain without providing additional value.

Recommendation:

Remove the redundant `if (!success)` block from the function. The function already performs required checks for successful transfers.

[G-01] Can use assembly for Null Address checks

Severity: Gas

**Location : **All Contracts

Description
You can use assembly for null address checks to save some gas .

Recommendation
Replace the high-level Solidity check with inline assembly to optimize gas usage. Inline assembly performs low-level operations, minimizing overhead.

```
modifier notZeroAddress(address _address) {
   assembly {
       if iszero(_address) {
           mstore(0x00, 0x0) // Store error selector (can be replaced with
a specific revert reason selector if necessary)
           revert(0x00, 0x20) // Revert with 32 bytes of error data
       }
   }
   _;
}
```

POC

```
   modifier isAllowedCollateral(address _token) {
       if (_token != address(i_empowerToken)) { //@audit-info so by this only empower token is allowed to be a collateral
           revert BorrowLend__CollateralNotAllowed();
       }
       _;
   }
```