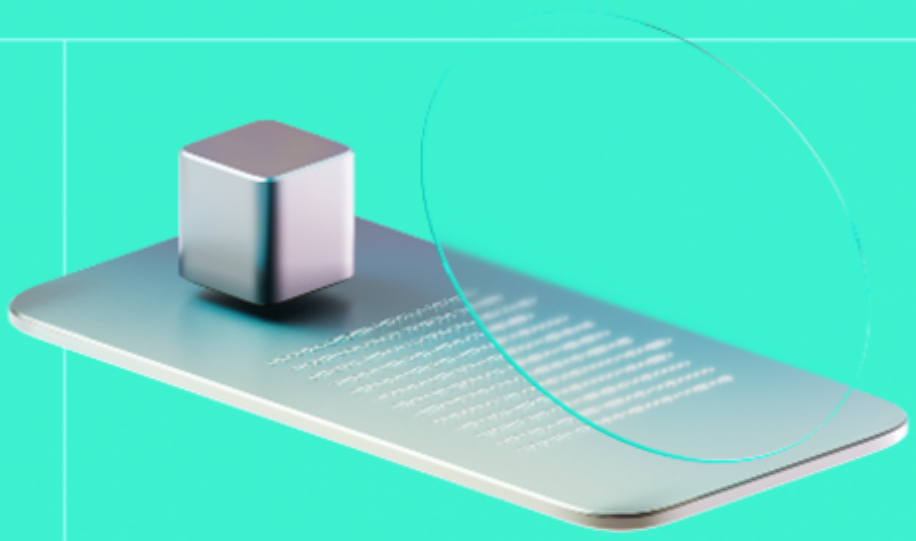
HACKEN

Smart Contract Code Review And Security Analysis Report

Customer: XPower

Date: 10/01/2024



We thank XPower for allowing us to conduct a Smart Contract Security Assessment. This document outlines our methodology, limitations, and results of the security assessment.

XPower aims to be a fairly distributed currency, leveraging CPU power to mine and distribute PoW coins.

Platform: EVM

Language: Solidity

Tags: ERC20; ERC1155; Staking

Timeline: 19/12/2023 - 10/01/2024

Methodology: https://hackenio.cc/sc_methodology

Last Review Scope

Repository	https://github.com/blackhan-software/xpower-hh/
Commit	272f69a

Audit Summary

10/10	9/10	80.31%	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.1/10

The system users should acknowledge all the risks summed up in the risks section of the report

6	6	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	1
High	2
Medium	0
Low	2

Vulnerability	Status
F-2023-0296 - Extended Validity of Block Hashes in XPower Contract Leads to Inflation Vulnerability	Fixed
F-2023-0321 - Inadequate Handling of NFT Transfers and Restaking in XPowerPpt.sol Impacts Reward Accumulation	Fixed
F-2023-0322 - Reward Accumulation Issue in MoeTreasury Contract Affecting Restaking and Transfers of XPowerPpt NFTs	Fixed
F-2023-0326 - Floating Pragma	Fixed
F-2024-0330 - Potential Forced Unstaking in unstake() and unstakeBatch() Functions During Migration	Fixed
F-2024-0333 - Forced Migration in Token Contracts	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for XPower
Audited By	Ivan Bondar
Approved By	Ataberk Yavuzer
Website	https://www.xpowermine.com/
Changelog	10/1/2024 - Final Report

Table to Contents

System Overview

XPower introduces XPOW tokens, leveraging a hybrid of Proof-of-Work (PoW) and Proof-of-Stake (PoS) on the Avalanche C-Chain. These tokens are minted through a PoW process, ensuring fair distribution without pre-mining or pre-minting. Anyone can mint XPOW tokens by submitting a valid nonce number.

The project combines the equitable distribution aspect of PoW with the efficient transaction capabilities of PoS. XPOW tokens can be transformed into NFTs, which are stakable for earning APower (APOW) tokens, representing matured XPower. Burning APower tokens releases the contained XPower based on the current conversion rate. Additionally, a portion of each mint contributes to the project's treasury, supporting its sustainability.

XPower aims to enhance the Avalanche ecosystem by merging the strengths of PoW and PoS, promoting a balanced and energy-efficient economy.

The files in the scope:

- **XPower.sol** - Manages XPOW tokens, mintable through proof-of-work without pre-mining. Tokens are minted by providing a valid nonce.
- **APower.sol** - Handles APOW tokens, mintable exclusively by the contract owner (MoeTreasury) at a controlled rate.
- **XPowerNft.sol** - Manages XPOW NFTs, mintable by depositing corresponding XPOW token amounts.
- **XPowerPpt.sol** - Manages staked XPowerNft tokens, allowing only the contract owner (NftTreasury) to mint and burn these tokens.
- **MoeTreasury.sol** - Treasury contract for minting APower tokens in exchange for staked XPowerNft tokens.
- **NftTreasury.sol** - Treasury contract for staking and unstaking XPowerNft tokens.
- **FeeTracker.sol** - Tracks fees using cumulative moving averages.
- **Migratable.sol** - Facilitates token migration from old contracts up to a specified deadline.
- **NftMigratable.sol** - Enables NFT migration from old contracts, including batch migration and manual sealing.
- **NftBase.sol** - Abstract base class for NFTs, incorporating multiple ERC1155 functionalities and extensions.
- **NftRoyalty.sol** - Manages NFT royalties, allowing changes in beneficiary with a default royalty fraction of 0.5%.
- **URIMalleable.sol** - Allows changing NFT URIs, with permanence conditions based on time, and supports contract-level metadata URI.
- **Supervised.sol** - Base contract for role management across the project.
- **Array.sol** - Library for array operations, including sorting and duplicate checking.
- **Constants.sol** - Stores constants used throughout the project's contracts.
- **Integrator.sol** - Library for integrating over arrays of tuples and calculating weighted arithmetic means.
- **Nft.sol** - Library for NFT-related operations, including ID composition and level denomination.
- **Polynomials.sol** - Library for evaluating values using linear functions defined by polynomial coefficients.
- **Power.sol** - Library for exponentiation operations, particularly raising numbers to specified powers.
- **Rpp.sol** - Library for rug pull protection, ensuring valid parameter changes and invocation frequencies.

Privileged roles

- **XPower.sol**:
  - **owner**: Can transfer ownership, renounce ownership. Co-mints extra XPower tokens for the treasury with each new minting.
  - **MOE_SEAL_ROLE** (from MoeMigratable): Grants the right to seal MOE immigration.
  - **MOE_SEAL_ADMIN_ROLE** (from MoeMigratable): Admin role for MOE_SEAL_ROLE, can set or revoke MOE_SEAL_ROLE.
- **APower.sol**:
  - **owner**: Only the owner can mint new APower tokens. Can also transfer and renounce ownership.
  - **SOV_SEAL_ROLE** (from SovMigratable): Grants the right to seal SOV immigration.
  - **SOV_SEAL_ADMIN_ROLE** (from SovMigratable): Admin role for SOV_SEAL_ROLE, can set or revoke SOV_SEAL_ROLE.
- **XPowerNft.sol**:
  - **NFT_SEAL_ROLE**: Grants the right to seal NFT immigration.
  - **NFT_SEAL_ADMIN_ROLE**: Admin role for NFT_SEAL_ROLE, can set or revoke NFT_SEAL_ROLE.
  - **NFT_ROYAL_ROLE**: Grants the right to set the NFT's default royalty beneficiary.
  - **NFT_ROYAL_ADMIN_ROLE**: Admin role for NFT_ROYAL_ROLE, can set or revoke NFT_ROYAL_ROLE.
  - **URI_DATA_ROLE**: Grants the right to change metadata URIs.

    - **URI_DATA_ADMIN_ROLE**: Admin role for URI_DATA_ROLE, can set or revoke URI_DATA_ROLE.
- **XPowerPpt.sol**:
    - **owner**: Only the owner can mint and burn XPowerPpt tokens. Can also transfer and renounce ownership.
    - **NFT_SEAL_ROLE**: Grants the right to seal NFT immigration.
    - **NFT_SEAL_ADMIN_ROLE**: Admin role for NFT_SEAL_ROLE, can set or revoke NFT_SEAL_ROLE.
    - **NFT_ROYAL_ROLE**: Grants the right to set the NFT's default royalty beneficiary.
    - **NFT_ROYAL_ADMIN_ROLE**: Admin role for NFT_ROYAL_ROLE, can set or revoke NFT_ROYAL_ROLE.
    - **URI_DATA_ROLE**: Grants the right to change metadata URIs.
    - **URI_DATA_ADMIN_ROLE**: Admin role for URI_DATA_ROLE, can set or revoke URI_DATA_ROLE.
- **MoeTreasury.sol**:
    - **APR_ROLE**: Grants the right to change APR parametrization.
    - **APR_ADMIN_ROLE**: Admin role for APR_ROLE, can set or revoke APR_ROLE.
    - **APB_ROLE**: Grants the right to change APB parametrization.
    - **APB_ADMIN_ROLE**: Admin role for APB_ROLE, can set or revoke APB_ROLE.

Executive Summary

This report presents an in-depth analysis and scoring of the Customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are detailed.
  - Project overview is detailed
  - All roles in the system are described.
  - Use cases are described and detailed.
  - For each contract, all futures are described.
  - All interactions are described.
- Technical description is limited.
  - Run instructions are provided.
  - Technical specification is provided.
  - The NatSpec documentation is sufficient.

Code quality

The total Code Quality score is **9** out of **10**.

- The development environment is configured.
- Solidity Style Guide violations.

Test coverage

Code coverage of the project is **80.31%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage are partially missed.
- Interactions by several users are tested.
- Not all branches are covered with tests.

Security score

Upon auditing, the code was found to contain **1** critical, **2** high, **0** medium, and **2** low severity issues, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the Customer's smart contract yields an overall score of **9.1**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- **Variation in APOW Token Supply Due to Claiming Frequency**: It is important to note that the actual minting amounts of APOW tokens may significantly deviate from initial projections due to the frequency of user claims. Particularly in the early stages of the token's existence, if users claim their rewards more frequently than the assumed annual rate, this behavior could lead to a substantially higher rate of token supply growth. Such an increase in the minting rate, driven by frequent claiming, presents a risk of accelerating the token supply beyond expected limits, potentially affecting the token's value and the ecosystem's economic balance.
- **Migration Risks with New Version Releases**: Users have the option to migrate their ERC20 and ERC1155 tokens to newer versions. However, migration is subject to deadlines and can be prematurely sealed by addresses with SEAL_ROLE, potentially impacting users' ability to migrate after the deadline.
- **Impact of Staking Different NFT Levels on Rewards**: The system auto-adjusts target reward rates to balance income across NFT levels (UNIT, KILO, MEGA). Staking higher-level NFTs can significantly alter rewards for all levels, leading to substantial changes in claimable rewards, especially for lower-level NFTs.
- **Administrative Control Over APR and APB Parameters**: The protocol incorporates roles like APR_ROLE and APB_ROLE, which enable the modification of APR and APB parameters. This structure is designed to allow flexible re-parametrization while preventing arbitrary changes. It ensures that participants have sufficient time to assess the reasonableness of target settings. Over time, the values tend to become more entrenched, making changes increasingly challenging. This mechanism balances flexibility with stability, but users should remain vigilant about potential adjustments.
- **Risk of Reward Reset on PPT Transfers**: Transferring PPTs (staked NFTs) without claiming accumulated rewards can lead to potential reward losses. Users are advised to claim rewards before any transfer.

Findings

Vulnerability Details

F-2023-0296 - Extended Validity of Block Hashes in XPower Contract Leads to Inflation Vulnerability - Critical

Description:

In the XPower contract, the `_recent` function plays a crucial role in determining the validity of block hashes for the purpose of minting tokens. This function is expected to ensure that a block hash is only used within a specific hourly interval post its generation. However, due to an incorrect implementation, the function fails to accurately enforce this time constraint.

```
function _recent(bytes32 blockHash) private view returns (bool) {
   return _timestamps[blockHash] > currentInterval();
}
```

The issue arises from the comparison of `_timestamps[blockHash]`, which stores the timestamp of the block hash from the first call of the `init` function in a new interval, with `currentInterval`, which returns the interval in hours. This comparison leads to block hashes being valid for an extended period, far exceeding the intended one-hour limit.

This vulnerability poses a significant risk of hidden inflation. Users can exploit the extended validity of block hashes to mint tokens for a prolonged period, leading to an increase in token supply beyond the intended rate. This inflation can devalue the XPOW tokens and undermine the integrity of the token economy.

Found In: bd660e1

Assets:

- XPower.sol

Status: Fixed

Classification

Severity: Critical

Impact: 5/5

Likelihood: 5/5

Recommendations

Recommendation:

Update the function to accurately compare timestamps, ensuring block hashes expire after the intended one-hour interval. The revised function should be as follows:

```
function _recent(bytes32 blockHash) private view returns (bool) {
   return _timestamps[blockHash] >= currentInterval() * (1 hours);
}
```

This modification ensures that a block hash is valid only for its specific hourly interval. It also addresses the edge case where a block hash set at the start of a new interval is immediately considered expired. By including equality in the comparison, block hashes generated at the interval boundary are correctly handled.

Remediation (Revised commit: a1e5641) : The issue with the `_recent` function in the XPower contract, which previously allowed block hashes to be valid for an extended period, was successfully addressed. The function was updated to ensure that block hashes expire after the intended one-hour interval, aligning with the original design intention and preventing hidden inflation risks.

Evidences

Inflation Vulnerability

Reproduce:

POC Steps:

- Initial Setup:
  - Deploy XPower Contract: Deploy the XPower contract in a test environment.
  - Initialize Contract: Invoke the init function to set the initial block hash and timestamp.
- Execute First Mint Transaction:
  - Mint Tokens: Use a valid nonce to mint tokens with the current block hash. Record the beneficiary's balance and the current interval.
- Simulate Time Passage:
  - Advance Time: Increase the simulation time by 24 hours to change the interval.
  - Verify Interval Change: Confirm the currentInterval function reflects the new interval.
- Execute Second Mint Transaction with Expired Block Hash:
  - Mint with Expired Hash: Attempt to mint tokens using the same block hash (now expired) with a new nonce.
  - Check for Success: If tokens are minted, it indicates the vulnerability where expired block hashes are still valid.
- Assess Impact:
  - Compare Balances: Observe the beneficiary's balance before and after the second minting to confirm additional tokens were minted using the expired block hash.

POC Code:

```
    it("Minting with expired blockHash", async function () {
        const contractAddress = moe.target;
        const toAddress = addresses[1];

        // Get the current interval and corresponding block hash
        let interval = await moe.currentInterval();
        const blockHash = await moe.blockHashOf(interval);
        const targetZeros = 1;

        // Solve the puzzle for the current interval and mint tokens
        let data = await solvePuzzle(contractAddress, toAddress, blockHash, targetZeros);
        await moe.mint(toAddress, blockHash, data);
        let balanceAfterFirstMint = await moe.balanceOf(toAddress);
        console.log("Balance after initial mint with correct interval block-hash: ", ethers.form
```

See more

Results:

```
  XPower
    Inflation Vulnerability
Solution found: Nonce = 288
Nonce Hash: 0x00e06f7d079e4c88731f20aa2e36e36c294cb059cdd54a54ce9e14a3df2b7577
Data for mint function: 0x0000000000000000000000000000000000000000000000000000000000000120
```

```
Balance after initial mint with correct interval block-hash:  3.375
Interval of first mint:  473198n
Interval of second mint:  473222n
Solution found: Nonce = 82898
Nonce Hash: 0x0000af839978af32bd07a812e28ef5b39481334927cbc970cd199cfc2a3a58f5
Data for mint function: 0x00000000000000000000000000000000000000000000000000000000000143d2
Balance after second mint with expired block-hash:  20.25
      ✔ Minting with expired blockHash (32024ms)


  1 passing (35s)
```

Files: XPower.Inflation.test.js

F-2023-0321 - Inadequate Handling of NFT Transfers and Restaking in XPowerPpt.sol Impacts Reward Accumulation - High

Description:

The XPowerPpt.sol contract, responsible for managing staked NFTs , exhibits a critical flaw in its handling of NFT transfers and subsequent restaking. The issue arises in the `_pushBurn` function, where the age of an NFT (`_age[account][nftId]`) is decremented using a calculation that can lead to negative values. This is particularly evident when a user transfers part or all of their staked NFTs and then restakes the same NFT ID. The `ageOf` function, which plays a pivotal role in reward calculation, returns zero in such scenarios, as it only considers positive age values. This behavior adversely affects the user's ability to accumulate rewards on newly staked NFTs.

Affected Code:

```
/** @return age seconds over all stakes */
function ageOf(
    address account,
    uint256 nftId
) external view returns (uint256) {
    int256 age = _age[account][nftId];
    if (age > 0) {
        uint256 balance = balanceOf(account, nftId);
        return balance * block.timestamp - uint256(age);
    }
    return 0;
}
    /** remember burn action */
function _pushBurn(address account, uint256 nftId, uint256 amount) private {
    _age[account][nftId] -= int256(amount * block.timestamp);
}
```

This flaw can lead to unfair penalization of users who transfer or unstake their XPowerPpt and later wish to stake again. It effectively nullifies the reward accumulation for the new stake, as the `ageOf` function returns zero, impacting the `rewardOf` calculation in the MoeTreasury.sol contract. Users may lose potential rewards due to this issue.

Found In: bd660e1

Assets:

- XPowerPpt.sol

Status: Fixed

Classification

Severity: High

Impact: 3/5

Likelihood: 5/5

Recommendations

Recommendation: To address this issue, the following changes are recommended:

- **Modify _pushBurn Logic:** Implement a revised `_pushBurn` function that proportionally adjusts the age based on the amount being burned relative to the total balance. This can prevent the age from becoming negative.
- **Change Age Mapping:** Alter the `_age` mapping from `mapping(address => mapping(uint256 => int256))` to `mapping(address => mapping(uint256 => uint256))`. This change simplifies the implementation by removing the need for casting between integer types and ensures that age values remain non-negative.
- **Update Age Calculation:** Amend the `ageOf` function to accurately reflect the age of NFTs post-transfer or restaking, ensuring that users can accumulate rewards fairly.

These modifications will enhance the contract's logic, ensuring equitable reward distribution and maintaining user confidence in the staking mechanism.

Proposed Code Change:

```
// In XPowerPpt.sol
function _pushBurn(address account, uint256 nftId, uint256 amount) private {
  uint256 currentBalance = balanceOf(account, nftId);
  require(currentBalance >= amount, "Insufficient Balance");
  _age[account][nftId] -= Math.mulDiv(_age[account][nftId], amount, currentBalance);
```

Remediation (Revised commit: a1e5641) : The issue in the XPowerPpt.sol contract, related to the handling of NFT transfers and subsequent restaking, was effectively resolved. The recommended modifications to the `_pushBurn` function and the `ageOf` calculation logic was implemented. These changes ensure fair reward accumulation for users who transfer or restake their XPowerPpt, maintaining the integrity of the staking mechanism.

Evidences

Inadequate Handling of NFT Transfers and Restaking

Reproduce:

POC Steps:

- Initial Setup: The owner mints and stakes NFTs.
- First Time Increase: The script simulates the passage of one year.
- First Age Check: The age of the staked NFTs is logged.
- Transfer of NFTs: All staked NFTs are transferred to another user.
- Second Age Check: The age of the staked NFTs for the original owner is checked again and expected to be 0 due to
- the transfer logic in the contract.
- Restaking by Original Owner: The original owner stakes a new portion of NFTs with the same ID.
- Second Time Increase: Time is increased by another year.
- Third Age Check: The age of the newly staked NFTs is checked.

POC Code:

```
it("Inadequate Handling of NFT Transfers and Restaking", async function () {
 const [owner, user] = [accounts[0], accounts[1]];
 await ppt.transferOwnership(nty);
 await mintToken(10000n * UNIT);
 await increaseAllowanceBy(10000n * UNIT, nft.target);
 const nft_id = await mintNft(0, 10000);
 await nft.setApprovalForAll(nty, true);
 // Owner stakes NFTs
 await nty.stake(owner, nft_id, 9000);
```

```
// Increase time by one year
await network.provider.send("evm_increaseTime", [YEAR]);
await network.provider.send("evm_mine");
// Check the current age of the staked NFT
const age = await ppt.ageOf(owner.address, nft_id);
console.log("Age of stake before transfer: ", age.toString());
// Owner transfers all staked NFTs to another user
await ppt.safeTransferFrom(owner.address, user.address, nft_id, 9000, "0x");
// Check the age after transfer
const ageAfterTransfer = await ppt.ageOf(owner.address, nft_id);
console.log("Age of stake after transfer: ", ageAfterTransfer.toString());
// Owner stakes a new portion of NFT with the same ID
await nty.stake(owner, nft_id, 100);
// Increase time by another year
await network.provider.send("evm_increaseTime", [YEAR]);
await network.provider.send("evm_mine");
// Check the age
```

See more

Results:

```
  NftTreasury
    stake
Age of stake before transfer:  2840184000000
Age of stake after transfer:  0
Age of the new stake after YEAR:  0
      ✓ Transfer NFT issue (318ms)
```

Files: F-2023-0321.test.js

F-2023-0322 - Reward Accumulation Issue in MoeTreasury Contract Affecting Restaking and Transfers of XPowerPpt NFTs - High

Description:

The XPowerPpt contract, which manages the staking of XPowerNft, exhibits a critical flaw in its interaction with the MoeTreasury.sol contract. When a user transfers a portion of their XPowerPpt (staked XPowerNft) to another user or unstakes the XPowerNft, the `_claimed[account][nftId]` in MoeTreasury remains unchanged. This leads to a situation where the user's ability to accumulate new rewards for the same NFT ID is hindered by the previously claimed rewards.

In the `claim` function of MoeTreasury.sol, the amount of claimable rewards is calculated and added to `_claimed[account][nftId]`. This value is crucial in determining future reward accumulation:

```
function claim(address account, uint256 nftId) external {
   uint256 amount = claimable(account, nftId);
   require(amount > 0, "nothing claimable");
   _claimed[account][nftId] += amount;
   ...
}
```

The `claimable` function computes the difference between the total rewards (`generalReward`) and the already claimed rewards (`claimedReward`). If `generalReward` is greater, it returns the difference; otherwise, it returns zero:

```
function claimable(address account, uint256 nftId) public view returns (uint256) {
   uint256 claimedReward = claimed(account, nftId);
   uint256 generalReward = rewardOf(account, nftId);
   if (generalReward > claimedReward) {
       return generalReward - claimedReward;
   }
   return 0;
}
```

This issue can significantly impact users ability to earn new rewards, especially in scenarios involving frequent adjustments to staking strategies or secondary market activities. It may disincentivize users from restaking or transferring XPowerPpt.

Found In: bd660e1

Assets:

- XPowerPpt.sol
- MoeTreasury.sol

Status: Fixed

Classification

Severity: High

Impact: 3/5

Likelihood: 5/5

Recommendations

Recommendation: To mitigate this issue, the following changes are recommended:

- **Modify _pushBurn in XPowerPpt:** Adjust the `_pushBurn` function to proportionally update the claimed rewards based on the new balance after a transfer or unstake.
- **Introduce refreshClaimed in MoeTreasury:** Add a new function to recalculate the claimed rewards, ensuring proportional adjustment post-transfer or unstake.
- **Ownership Adjustment:** Make XPowerPpt the owner of MoeTreasury to enforce access control, allowing only XPowerPpt to update claimed rewards.

Proposed Code Changes:

```
// In XPowerPpt.sol
function _pushBurn(address account, uint256 nftId, uint256 amount) private {
   uint256 balance = balanceOf(account, nftId);
   require(balance >= amount, "ERC1155: insufficient balance for transfer");
   _mty.refreshClaimed(account, nftId, balance, balance - amount);
   _age[account][nftId] -= Math.mulDiv(_age[account][nftId], amount, balance);
}
// In MoeTreasury.sol
function refreshClaimed(address account, uint256 nftId, uint256 balanceOld, uint256 balanceNew)
   _claimed[account][nftId] = Math.mulDiv(_claimed[account][nftId], balanceNew, balanceOld);
}
```

Remediation (Revised commit: a1e5641) : The critical issue affecting reward accumulation in the XPowerPpt and MoeTreasury contracts was successfully addressed. The implementation of the proposed changes in the `_pushBurn` function of XPowerPpt and the introduction of the `refreshClaimed` function in MoeTreasury ensure fair and accurate reward calculations post-transfer or unstake of XPowerPpt.

Evidences

Reward Accumulation Issue in MoeTreasury Contract

Reproduce:

POC Steps:

- Stake NFTs: The owner stakes 9,000 units of the NFT (nft_id).
- Time Elapse: Simulate the passage of one year.
- Claim Rewards: The owner claims accumulated rewards for the staked NFTs.
- Transfer Staked NFTs: The owner transfers all 9,000 staked NFTs (XPowerPpt) to the user.
- Restake NFTs: The owner stakes an additional 1,000 units of the same NFT ID (nft_id).
- Time Elapse: Again, simulate the passage of another year.
- Check Age and Rewards: Assess the age and claimable rewards for the owner and user for the NFT ID after the second year.

POC Code:

```
it("Reward Accumulation Issue in MoeTreasury Contract", async function () {
  const [owner, user] = [accounts[0], accounts[1]];
  await ppt.transferOwnership(nty);
  await sov.transferOwnership(mty);
  await mintToken(100000n * UNIT);
  await increaseAllowanceBy(100000n * UNIT, nft.target);
  const nft_id = await mintNft(0, 18000);
  await nft.setApprovalForAll(nty, true);
  // Owner stakes NFTs
  await nty.stake(owner, nft_id, 9000);
  // Increase time by one year
  await network.provider.send("evm_increaseTime", [YEAR]);
```

```
    await network.provider.send("evm_mine");
    // Check the current age of the staked NFT
    const age = await ppt.ageOf(owner.address, nft_id);
    console.log("Age of stake before transfer: ", age.toString());
    // Check the current claimable for staked NFTs
    claimable = await mty.claimable(owner.address, nft_id)
    console.log("claimable owner after 1 Year nft_level_0: ", (ethers.formatEther(claimable)) + "e
    // Owner transfers all staked NFTs to another user
    await mty.claim(owner, nft_id)
    await ppt.safeTransferFrom(owner.address, user.address, nft_id, 9000, "0x");
    // Check the age after transfer
    const ageAfterTransfer = await ppt.ageOf(owner.address, nft_id);
    console.log("Age of stake after transfer: ", ageAfterTransfer.toString());

    // Owner stakes a new port
```

See more

Results:

```
  NftTreasury
    stake
Age of stake before transfer:  284018400000
claimable owner after 1 Year nft_level_0:  281.28456e18
Age of stake after transfer:  0
Age of the new stake after YEAR:  0
claimable owner after 1 Year nft_level_0:  0.0e18
claimable user after 1 Year nft_level_0:  293.400018594569929272e18
      ✔ Reward Accumulation Issue in MoeTreasury Contract (679ms)
```

Files: F-2023-0322.test.js

F-2024-0330 - Potential Forced Unstaking in unstake() and unstakeBatch() Functions During Migration - Low

Description:

In the NftTreasury, the `unstake` and `unstakeBatch` functions in a contract are designed to allow users to unstake their NFTs. However, a potential vulnerability arises due to the conditional check `|| _nft.migratable()` within the `require` statements of these functions. This condition could allow for forced unstaking of NFTs during the migration period, which is contrary to the voluntary nature of migration as outlined in the project's documentation.

The `require` statements are intended to ensure that only the token owner or an approved entity can initiate the unstaking process. However, the inclusion of `|| _nft.migratable()` means that if the NFTs are marked as migratable (which happens during a migration period), any caller can trigger the unstaking process, potentially leading to unauthorized or unintended unstaking of NFTs.

Affected Code:

```
/** unstake NFT */
function unstake(address account, uint256 nftId, uint256 amount) external {
    require(
        (account == msg.sender || approvedUnstake(account, msg.sender)) || _nft.migratable()
        "caller is not token owner or approved"
    );
    _ppt.burn(account, nftId, amount);
    _nft.safeTransferFrom(address(this), account, nftId, amount, "");
    emit Unstake(account, nftId, amount);
    _mty.refreshRates(false);
}
```

The primary impact of this vulnerability is the risk of unauthorized unstaking of NFTs during the migration period. This could lead to disruption for users who did not intend to unstake their assets, potentially affecting their staking strategies and rewards. It also undermines the voluntary nature of the migration process.

Found In: bd660e1

Assets:

- NftTreasury.sol

Status: Fixed

Classification

Severity: Low

Impact: 3/5

Likelihood: 2/5

Recommendations

Recommendation:

To mitigate this vulnerability and align the contract behavior with the intended voluntary migration process, the following changes are recommended:

- Remove the `|| _nft.migratable()` condition from the `require` statements in both the `unstake` and `unstakeBatch` functions.

- Ensure that the unstaking process can only be initiated by the NFT owner or an approved entity, regardless of the migration status of the NFTs.

Proposed Code Changes: Modify the `require` statements in the `unstake` and `unstakeBatch` functions as follows:

```
// In the unstake function
require(
   account == msg.sender || approvedUnstake(account, msg.sender),
   "caller is not token owner or approved"
);
// In the unstakeBatch function
require(
   account == msg.sender || approvedUnstake(account, msg.sender),
   "caller is not token owner or approved"
);
```

Remediation (Revised commit: a1e5641) : The conditional check `|| _nft.migratable()` was removed, ensuring that only the NFT owner or an approved entity can initiate the unstaking process, thereby upholding the voluntary nature of the migration process.

F-2024-0333 - Forced Migration in Token Contracts - Low

Description:

The contracts in question (NftMigratable.sol, Migratable.sol, and SovMigratable.sol) are designed to facilitate the migration of tokens (both ERC1155 and ERC20) from older versions to newer ones. A observation in these contracts is the lack of restrictions on who can initiate the migration process. Specifically, the `migrateFrom` and `migrateFromBatch` functions in these contracts can be called with any `account` parameter, potentially allowing one user to initiate the migration of tokens belonging to another user.

Affected Code from NftMigratable.sol:

```
function migrateFrom(address account, uint256 nftId, uint256 amount, uint256[] memory index)
    _migrateFrom(account, nftId, amount, index);
}

/** migrate amount of ERC1155 */
function _migrateFrom(address account, uint256 nftId, uint256 amount, uint256[] memory index
    require(_deadlineBy >= block.timestamp, "deadline passed");
    _burnFrom(account, nftId, amount, index);
    _mint(account, nftId, amount, "");
}
```

This design raises a concern: if a user has approved a new version of a contract to access their tokens (old versions of ERC20 APower tokens, old versions of ERC20 Xpower tokens, old versions of NFTs XPowerNft.sol or XPowerPpt.sol), another user could potentially force the migration of those tokens without the token owner's explicit consent. This scenario contradicts the voluntary nature of migration as outlined in the project's documentation.

The impact of this vulnerability is twofold:

- Unauthorized Migration: Token owners may find their tokens migrated to a new contract without their knowledge or consent. This could lead to confusion and a lack of trust in the token system.
- Potential for Griefing: Malicious actors could exploit this vulnerability to disrupt the token holdings of unsuspecting users, leading to a negative user experience and potential reputational damage to the project.

Found In: bd660e1

Assets:

- NftMigratable.sol
- Migratable.sol

Status: Fixed

Classification

Severity: Low

Impact: 3/5

Likelihood: 2/5

Recommendations

Recommendation:

To mitigate this vulnerability and align the migration process with its intended voluntary nature, the following changes are recommended:

- Implement Access Controls: Modify the `migrateFrom` and `migrateFromBatch` functions to check that the caller is either the token owner or an explicitly authorized migrator. This ensures that only the token owner or a party authorized by them can initiate the migration process.
- Introduce a Consent Mechanism: Consider implementing a function that allows token owners to explicitly consent to migration. This function could set a flag indicating the owner's approval for migration, which the migrateFrom functions would then check.
- Enhance User Communication: Clearly communicate to users the process and requirements for migration, including any actions they need to take to authorize or initiate the migration.

Remediation (Revised commit: a1e5641) : The issue related to unrestricted migration initiation in the NftMigratable.sol, Migratable.sol, and SovMigratable.sol contracts was effectively resolved. Access controls was implemented in the `migrateFrom` and `migrateFromBatch` functions, ensuring that only the token owner or an authorized migrator can initiate the migration process, aligning with the intended voluntary nature of migration.

F-2023-0326 - Floating Pragma - Info

Description:

The project uses floating pragmas `^0.8.20`.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version, which may include bugs that affect the system negatively.

Found In: bd660e1

Assets:

- NftTreasury.sol
- MoeTreasury.sol
- XPowerPpt.sol
- XPower.sol
- XPowerNft.sol
- NftMigratable.sol
- FeeTracker.sol
- Migratable.sol
- Supervised.sol
- URIMalleable.sol
- Array.sol
- Constants.sol
- Integrator.sol
- Nft.sol
- Polynomials.sol
- Power.sol
- Rpp.sol
- APower.sol
- NftBase.sol
- NftRoyalty.sol

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation:

Replace the floating pragma `^0.8.20` with a fixed version, such as `pragma solidity 0.8.20`. This explicitly sets the compiler version to **0.8.20**, aligning the development, testing, and deployment environments.

Remediation (Revised commit: a1e5641) : Issue was resolved by setting a fixed compiler version (`pragma solidity 0.8.20`).

Observation Details

F-2023-0320 - Inefficient NFT ID Migration Logic in NftMigratable.sol Leads to Unnecessary Gas Usage - Info

Description:

The NftMigratable.sol contract employs a complex method for calculating `tryId` in the `_burnFrom` function. This function adds `Nft.eonOf(Nft.yearOf(nftId)) * 2` to the original `nftId`. Given the structure of `nftId` as `yearLL` (where `year` is the year of creation starting from 2021 and `LL` is a level number divisible by 3), and considering that the `eonOf` function consistently returns 1,000,000 for years 2021-9999, the `tryId` calculation effectively adds 2,000,000 to any `nftId` within this range. This approach, while functional, results in unnecessary complexity and increased Gas costs due to the redundant use of the `eonOf()` function.

This inefficient calculation method leads to higher Gas costs during NFT migration. The additional computational steps do not contribute to functional improvements or security enhancements, thus representing an area for optimization.

Found In: bd660e1

Assets:

- NftMigratable.sol

Status: Fixed

Recommendations

Recommendation:

To optimize the contract and reduce Gas costs, the following changes are recommended:

- **Introduce a Constant:** Define a constant for the value 2,000,000 in the contract, with NatSpec documentation explaining its historical significance in the context of NFT ID evolution.
- **Remove Redundant Function:** If the `eonOf` function is solely used for this calculation, consider removing it to simplify the codebase.
- **Simplify tryId Calculation:** Amend the `tryId` calculation to `tryId = nftId + 2_000_000;`, directly adding the constant to the `nftId`. This change will maintain the intended functionality while reducing gas costs and improving code clarity.

These modifications will streamline the migration logic, enhance efficiency, and reduce transaction costs for users interacting with the contract.

Remediation (Revised commit: a1e5641) : The complexity in the `tryId` calculation within the NftMigratable.sol contract's `_burnFrom` function was effectively streamlined. The recommended changes, including the introduction of a constant for 2,000,000 and the removal of the redundant `eonOf` function, was implemented.

F-2023-0323 - Inefficient Handling of Zero Amount in XPowerPpt NFT Transfers and Unstakes - Info

Description:

The `_pushBurn` function within the XPowerPpt contract is integral to the process of burning NFTs and updating claimed rewards. This function is invoked during the transfer of XPowerPpt NFTs or during the unstaking process.

The function lacks a zero amount check. This can lead to redundant operations when the function is called with an amount of zero, which might occur due to user input errors or specific unstaking scenarios.

Affected Code:

```
function _pushBurn(address account, uint256 nftId, uint256 amount) private {
    uint256 balance = balanceOf(account, nftId);
    require(
        balance >= amount,
        "ERC1155: insufficient balance for transfer"
    );
    _mty.refreshClaimed(account, nftId, balance, balance - amount);
    _age[account][nftId] -= Math.mulDiv(
        _age[account][nftId], amount, balance
    );
}
```

This issue, while not directly compromising security, leads to unnecessary computational steps and gas consumption.

Found In: d3b0271

Assets:

- NftTreasury.sol
- XPowerPpt.sol

Status: Fixed

Recommendations

Recommendation:

To optimize the contract and prevent unnecessary operations, the following update is recommended:

- **Implement a Zero Amount Check in the Require Statement:** Introduce a condition within the `require` statement to ensure that the `amount` is greater than zero. This will prevent the function from performing unnecessary actions when the amount is zero.

Proposed Code Adjustment:

```
function _pushBurn(address account, uint256 nftId, uint256 amount) private {
    require(
        amount > 0 && balanceOf(account, nftId) >= amount,
        "ERC1155: invalid amount or insufficient balance for transfer"
    );
    _mty.refreshClaimed(account, nftId, balanceOf(account, nftId), balanceOf(account, nftId) - am
    _age[account][nftId] -= Math.mulDiv(
        _age[account][nftId], amount, balanceOf(account, nftId)
```

```
    );
}
```

Remediation (Revised commit: a1e5641): The `_pushBurn` function in the XPowerPpt contract was updated to include a zero amount check. This enhancement prevents redundant operations and unnecessary gas consumption when the function is called with an amount of zero.

F-2023-0324 - Lack of Zero Amount Check in NFT Staking Functions - Info

Description:

In the `stake` and `stakeBatch` functions of a NftTreasury contract, there is a notable absence of checks to ensure that the amount of NFTs being staked is greater than zero. This oversight can lead to unnecessary execution of these functions when the amount is zero, potentially causing redundant Gas expenditure and affecting the contract's efficiency.

Affected Code:

```
function stake(address account, uint256 nftId, uint256 amount) external {
    require(
        (account == msg.sender || approvedStake(account, msg.sender)) && !_nft.migratable(),
        "caller is not token owner or approved"
    );
    _nft.safeTransferFrom(account, address(this), nftId, amount, "");
    _ppt.mint(account, nftId, amount);
    emit Stake(account, nftId, amount);
    _mty.refreshRates(false);
}
function stakeBatch(
    address account,
    uint256[] memory nftIds,
    uint256[] memory amounts
) external {
    require(
        (account == msg.sender || approvedStake(account, msg.sender)) && !_nft.migratable(),
        "caller is not token owner or approved"
    );
    _nft.safeBatchTransferFrom(account, address(this), nftIds, amounts, "");
    _ppt.mintBatch(account, nftIds, amounts);
    emit StakeBatch(account, nftIds, amounts);
    _mty.refreshRates(false);
}
```

The lack of a zero amount check can lead to situations where users inadvertently execute staking transactions with zero amounts, resulting in unnecessary Gas costs and potential confusion.

Found In: bd660e1

Assets:

- NftTreasury.sol

Status: Fixed

Recommendations

Recommendation:

It is recommended to add a check for zero amounts in both the `stake` and `stakeBatch` functions. This will prevent the functions from executing when the staking amount is zero.

Remediation (Revised commit: a1e5641) : The issue regarding the absence of zero amount checks in the `stake` and `stakeBatch` functions of the NftTreasury contract was successfully addressed. The functions now include validations to prevent execution with zero amounts, enhancing efficiency and reducing unnecessary Gas costs.

F-2023-0327 - Optimization and Enhancement of claimBatch() Function in MoeTreasury - Info

Description:

The `claimBatch` function in the smart contract is designed to process multiple NFT IDs for claiming rewards. However, the current implementation has two primary issues:

- **Reversion on Zero Rewards:** The function iterates over an array of NFT IDs and checks the claimable amount for each ID. The `require(amounts[i] > 0, "nothing claimable");` statement causes the entire batch process to revert if any NFT ID within the batch has a zero claimable amount. This behavior is inefficient as it prevents the processing of valid, non-zero claims within the same batch, requiring users to manually filter out NFT IDs with zero rewards.
- **Inefficient Resource Utilization:** The function makes repeated calls to `_sov.mintable(amounts[i])` and `_moe.increaseAllowance(address(_sov), _sov.wrappable(amounts[i]));` within the loop. This approach is inefficient as it leads to multiple state changes and higher gas costs.

Affected Code:

```
 /** claim APower tokens */
function claimBatch(address account, uint256[] memory nftIds) external {
    require(Array.unique(nftIds), "unsorted or duplicate ids");
    uint256[] memory amounts = claimableBatch(account, nftIds);
    for (uint256 i = 0; i < nftIds.length; i++) {
        require(amounts[i] > 0, "nothing claimable");
        _claimed[account][nftIds[i]] += amounts[i];
        _minted[account][nftIds[i]] += _sov.mintable(amounts[i]);
        _moe.increaseAllowance(address(_sov), _sov.wrappable(amounts[i]));
        _sov.mint(account, amounts[i]);
    }
    emit ClaimBatch(account, nftIds, amounts);
}
```

The current implementation can lead to unnecessary gas expenditure and may prevent users from claiming rewards efficiently. In scenarios where the batch contains NFT IDs with zero rewards, the entire transaction fails, requiring users to manually filter out such IDs. Additionally, the repeated state changes within the loop increase transaction costs and reduce contract efficiency.

Found In: bd660e1

Assets:

- MoeTreasury.sol

Status: Accepted

Recommendations

Recommendation:

To optimize the **claimBatch** function and enhance its efficiency:

- **Skip Zero Rewards:** Replace the `require` statement with an `if` condition to continue the loop if the claimable amount is zero. This allows the function to process only NFT IDs with positive rewards, skipping those with zero rewards.
- **Aggregate Calculations:** Introduce local variables to accumulate the total amounts for `_sov.wrappable(amounts[i])` and the total mintable amounts. Perform the `increaseAllowance` and `mint` operations once after the loop, using these aggregated totals.

Proposed Code Changes:

```
function claimBatch(address account, uint256[] memory nftIds) external {
   require(Array.unique(nftIds), "unsorted or duplicate ids");
   uint256[] memory amounts = claimableBatch(account, nftIds);
   uint256 totalWrappable = 0;
   uint256 totalMintable = 0;
   for (uint256 i = 0; i < nftIds.length; i++) {
       if (amounts[i] > 0) {
           _claimed[account][nftIds[i]] += amounts[i];
           totalMintable += _sov.mintable(amounts[i]);
           totalWrappable += _sov.wrappable(amounts[i]);
       }
   }
   if (totalWrappable > 0) {
       _moe.increaseAllowance(address(_sov), totalWrappable);
       _sov.mint(account, totalMintable);
   }
   emit ClaimBatch(account, nftIds, amounts);
}
```

Remediation (Revised commit: a1e5641) : The `claimBatch` function in the smart contract was modified to include an initial check of all claimable amounts in a separate array. This adjustment ensures that the function only proceeds with state updates and calls to `increaseAllowance` and `mint` if the amounts are valid. However, the function still makes multiple calls to `increaseAllowance` and `mint` and will revert if any NFT ID in the batch has a zero claimable amount.

F-2023-0328 - Inefficient Gas Usage in burnBatch() Function Due to Redundant Operations of the XPowerNft.sol contract - Info

Description:

The `burnBatch` function in the smart contract is designed to burn multiple NFTs and redeem corresponding MOE tokens for each. The function iterates over arrays of NFT IDs (`ids`) and their respective amounts (`amounts`), performing a redeemability check and a MOE token transfer for each NFT ID in the batch. However, the current implementation has two main inefficiencies:

- **Redundant Redeemability Checks:** The redeemability check (`require(_redeemable(ids[i]), "irredeemable issue");`) is performed inside the loop for each NFT ID. This approach can lead to unnecessary gas consumption if an irredeemable NFT ID is included in the batch, as the function will revert after executing the burn operation for previous IDs.
- **Multiple Transfer Calls:** The function executes a MOE token transfer for each NFT ID within the loop. This results in multiple transfer calls, which is less efficient compared to aggregating the total MOE amount and performing a single transfer.

Affected Code:

```
 /** batch-burn NFTs */
function burnBatch(
    address account,
    uint256[] memory ids,
    uint256[] memory amounts
) public override {
    super.burnBatch(account, ids, amounts);
    _redeemToBatch(account, ids, amounts);
}
function _redeemToBatch(
    address account,
    uint256[] memory ids,
    uint256[] memory amounts
) private {
    for (uint256 i = 0; i < ids.length; i++) {
        require(_redeemable(ids[i]), "irredeemable issue");
        uint256 moeAmount = amounts[i] * denominationOf(levelOf(ids[i]));
        _moe.transfer(account, moeAmount * 10 ** _moe.decimals());
    }
}
```

The inefficiencies in the **burnBatch** function can lead to higher gas costs for users, especially when the batch contains a large number of NFTs. In cases where an irredeemable NFT ID is included, users may experience failed transactions and lose gas fees due to the lack of upfront validation for redeemability. Additionally, multiple transfer calls within the loop increase the overall gas cost of the transaction.

Found In: bd660e1

Assets:

- XPowerNft.sol

Status: Fixed

Recommendations

Recommendation:

The **burnBatch** function and the **_redeemToBatch** function can be updated as follows to optimize for Gas efficiency:

- **burnBatch Function:**
  - Move the redeemability check to the beginning of the function.
  - Call the **_redeemToBatch** function after performing the burn operation.

```
function burnBatch(
    address account,
    uint256[] memory ids,
    uint256[] memory amounts
) public override {
    for (uint256 i = 0; i < ids.length; i++) {
        require(_redeemable(ids[i]), "irredeemable issue");
    }
    super.burnBatch(account, ids, amounts);
    _redeemToBatch(account, ids, amounts);
}
```

- **redeemToBatch Function:**
  - Aggregate the total MOE amount within the loop.
  - Perform a single MOE transfer after the loop.

```
function _redeemToBatch(
    address account,
    uint256[] memory ids,
    uint256[] memory amounts
) private {
    uint256 totalMoeAmount = 0;
    for (uint256 i = 0; i < ids.length; i++) {
        totalMoeAmount += amounts[i] * denominationOf(levelOf(ids[i]));
    }
    _moe.transfer(account, totalMoeAmount * 10 ** _moe.decimals());
}
```

These changes ensure that the redeemability of all NFTs is checked before any are burned, preventing unnecessary Gas expenditure on failed transactions. Additionally, aggregating the total MOE amount for a single transfer after the loop reduces the number of transfer calls, further optimizing Gas usage.

Remediation (Revised commit: a1e5641) : The `burnBatch` function in the smart contract was updated to optimize gas efficiency. The redeemability of all NFTs is now checked upfront before any burning occurs, preventing unnecessary gas expenditure on failed transactions. Additionally, the total `MOE` amount is aggregated for a single transfer after the loop, reducing the number of `transfer` calls and further optimizing gas usage.

F-2024-0329 - Variable Shadowing in Inherited Contracts Leads to Ambiguity in APower and SovMigratable - Info

Description:

In the Solidity smart contract system, variable shadowing occurs when two variables in an inheritance hierarchy share the same name. This issue is observed in the APower contract, which inherits from SovMigratable. Both contracts declare a private variable named `_moe`, but they refer to different types: `XPower` in APower and `MoeMigratable` in SovMigratable. This overlapping naming convention can lead to confusion and potential bugs, as it obscures which variable is being accessed or modified in various parts of the code.

The primary impact of this issue is the potential for logical errors and bugs in the contract's execution. These errors might not be immediately apparent but can manifest in unexpected behavior, especially when the contracts are updated or extended in the future. The clarity and maintainability of the code are also significantly reduced, making it harder for developers to understand and safely modify the contract.

Found In: bd660e1

Status: Fixed

Recommendations

Recommendation:

To resolve this issue and enhance code clarity, the following changes are recommended:

- Rename one of the `_moe` variables to a more descriptive and unique identifier. For example, in the APower contract, consider renaming `_moe` to `_xPowerMoe`.
- Update all references to the renamed variable in the APower contract to reflect this change.
- Ensure that similar naming conflicts are checked for and resolved throughout the codebase to prevent similar issues in other parts of the system.

Remediation (Revised commit: a1e5641) : The `_moe` variable in SovMigratable was renamed to `_moeMigratable`, eliminating the naming conflict and enhancing code clarity. All references to this variable was updated accordingly, ensuring improved maintainability and reducing the potential for logical errors in the contract's execution.

F-2024-0331 - Redundant Assert Statements in init() Function of XPower Contract - Info

Description:

In the `init` function of a XPower smart contract, there are three `assert` statements used to validate conditions that are inherently true in the context of blockchain operations. The function is designed to initialize certain values based on the block hash and timestamp. The relevant code snippet is as follows:

```
function init() external {
    uint256 interval = currentInterval();
    assert(interval > 0);
    if (uint256(_blockHashes[interval]) == 0) {
        bytes32 blockHash = blockhash(block.number - 1);
        assert(blockHash > 0);
        uint256 timestamp = block.timestamp;
        assert(timestamp > 0);
        _blockHashes[interval] = blockHash;
        _timestamps[blockHash] = timestamp;
        emit Init(blockHash, timestamp);
    } else {
        bytes32 blockHash = _blockHashes[interval];
        uint256 timestamp = _timestamps[blockHash];
        emit Init(blockHash, timestamp);
    }
}
```

The `assert` statements in question are:

- `assert(interval > 0)`: This checks if the current interval, derived from the block timestamp, is greater than zero. Given that block timestamps on the Avalanche C-Chain are always positive, this condition will always hold true.
- `assert(blockHash > 0)`: This checks if the hash of the previous block (`block.number - 1`) is non-zero. In blockchain operations, block hashes are always non-zero values.
- `assert(timestamp > 0)`: This checks if the `block.timestamp` is positive, which is always the case in blockchain environments.

While these assert statements do not pose a direct security risk, they contribute to unnecessary gas consumption and code complexity.

Found In: bd660e1

Assets:

- XPower.sol

Status: Fixed

Recommendations

Recommendation:

To enhance the efficiency of the `init` function, the following changes are recommended:

Remove Redundant Asserts:

- Eliminate the `assert(interval > 0)`, `assert(blockHash > 0)`, and `assert(timestamp > 0)` statements.
- These conditions are inherently true in the context of Avalanche C-Chain operations, making the asserts superfluous.

Code Simplification:

- Simplify the `init` function by removing these asserts, which will reduce gas costs and improve the readability of the code.

Remediation (Revised commit: a1e5641) : The redundant `assert` statements in the `init` function of the XPower smart contract were successfully removed. This update streamlines the function, reducing unnecessary gas consumption and enhancing code readability.

F-2024-0332 - Lack of Events in Key Functions - Info

Description:

In the smart contract ecosystem, events are crucial for tracking changes and providing transparency. However, in several key functions across multiple contracts, events are notably absent.

The functions in question are:

- MoeTreasury: `setAPR`, `setAPB`
- Migratable: `_seal`
- NftMigratable: `seal`, `sealAll`
- NftRoyalty: `setRoyal`
- XPowerPpt: `initialize`

The absence of events in these key functions can lead to a lack of transparency and hinder effective monitoring. This can be particularly concerning in scenarios where contract administrators or users need to verify that certain actions have been performed correctly.

Found In: d3b0271

Assets:

- XPowerPpt.sol
- MoeTreasury.sol
- Migratable.sol
- NftMigratable.sol
- NftRoyalty.sol

Status: Fixed

Recommendations

Recommendation:

To address this issue, it is recommended to add appropriate event emissions in each of these key functions. The events should be designed to log relevant information that reflects the changes made by the function calls. Specifically:

Add Event Emissions:

- For each of the functions mentioned (`setAPR`, `setAPB`, `_seal`, `seal`, `sealAll`, `setRoyal`,`initialize`), define a corresponding event that logs the key parameters and outcomes of the function call.
- Ensure that the event names are descriptive and consistent with the function's purpose.

Implement Detailed Logging:

- The events should include parameters that provide a clear and comprehensive log of the actions taken within the function. This may include identifiers, new settings, timestamps, and any other relevant data.

Remediation (Revised commit: a1e5641) : The issue regarding the absence of events in key functions across multiple contracts was addressed. Events were added, enhancing transparency and enabling effective monitoring of contract actions.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/blackhan-software/xpower-hh
Commit	bd660e12ee394e736d1a65f4d2282ed16df21756
Whitepaper	N/A
Requirements	https://www.xpowermine.com/about
Technical Requirements	https://www.xpowermine.com/about

Contracts in Scope

./contracts/base/FeeTracker.sol

./contracts/base/Migratable.sol

./contracts/base/NftBase.sol

./contracts/base/NftMigratable.sol

./contracts/base/NftRoyalty.sol

./contracts/base/Supervised.sol

./contracts/base/URIMalleable.sol

./contracts/libs/Array.sol

./contracts/libs/Constants.sol

./contracts/libs/Integrator.sol

./contracts/libs/Nft.sol

./contracts/libs/Polynomials.sol

./contracts/libs/Power.sol

./contracts/libs/Rpp.sol

./contracts/APower.sol

./contracts/MoeTreasury.sol

./contracts/NftTreasury.sol

./contracts/XPower.sol

./contracts/XPowerNft.sol

./contracts/XPowerPpt.sol