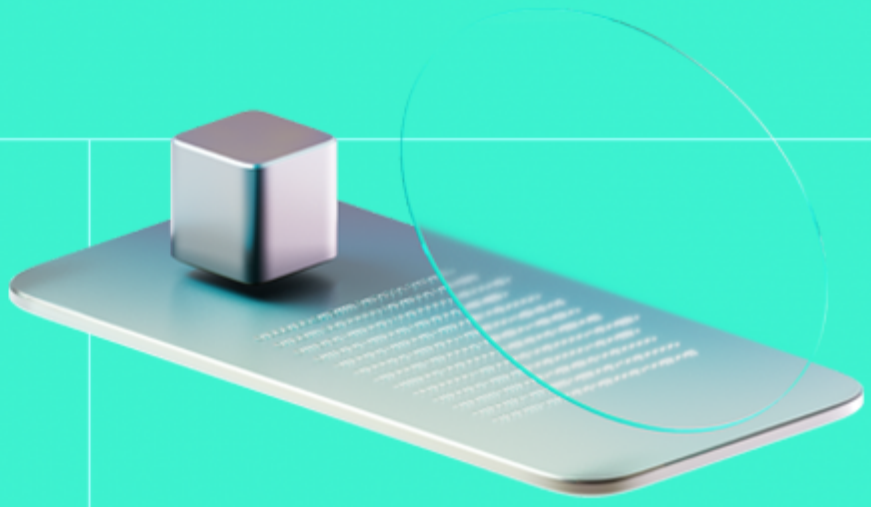
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Nexpace Limited

Date: 15/04/2025

We express our gratitude to the Nexpace Limited team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

The MapleStory Universe (MSU) is a game universe that combines the globally recognized MapleStory IP with blockchain.

This report consolidates other audit reports for Nexpace Limited organization. Please check the **Appendix 2. Scope** section for more details. The final audit was performed for the `feature/25-hacken-audit` branch with the following scope.

Significant changes (full audit):

- ItemIssuance/ItemIssuance.sol (173 LoC)
- NXPCDistributor.sol (280 LoC)

Minor changes:

- contracts/Bridge/DestinationChain/DestinationBridge.sol (only name convention changes)
- contracts/Bridge/DestinationChain/ERC1155BridgeToken.sol (Destination Bridge import removal)
- contracts/Bridge/DestinationChain/ERC20BridgeToken.sol (Destination Bridge import removal)
- contracts/Bridge/DestinationChain/ERC721BridgeToken.sol (Destination Bridge import removal)
- contracts/Bridge/Teller/Teller.sol (import order change)
- contracts/Creator/CreatorWallet/CreatorWalletLogicUpgradeable.sol (function parameter renaming)
- contracts/NXPC/NXPCAmountManager.sol (only name convention changes)

Document

Name	Smart Contract Code Review and Security Analysis Report for Nexpace Limited
Audited By	Ataberk Yavuzer
Approved By	Ivan Bondar
Website	https://msu.io/
Changelog	04/04/2025 - Preliminary Report (Merge)
	15/04/2025 - Final Report
Platform	Henesys (Avalanche L1)
Language	Solidity
Tags	Marketplace, ERC721, Proxy, GameFi, Claims, Merkle Tree
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository #1	https://github.com/NEXPACE-Limited/min-proxy
Commit #1	1af3f49415c22bf2128fdc2b832cb375d6cd9f1e
Repository #2	https://github.com/NEXPACE-Limited/util-contracts
Commit #2	0e51eaa85e5b2f9dc391d87a04c5a84936db8aa7
Repository #3	https://github.com/NEXPACE-Limited/nexpace-contracts
Commit #3	b65662b9e3db5fc7ca20ad8ed860ec8b6920def1
Repository #4	https://github.com/NEXPACE-Limited/marketplace-contract
Commit #4	7033807639afb6d9b89feedc7113ad53290f74bd
Repository #5	https://github.com/NEXPACE-Limited/maple-contract
Commit #5	5f83de2960f79ffb0fbc21641aaf98ab31b1712d
Repository #6	https://github.com/NEXPACE-Limited/msn-contract
Commit #6	08809acd7a462b8c461c35fad4fbc9fd7eb494cf
Repository #7	https://github.com/NEXPACE-Limited/multisig-contract
Commit #7	91526df0a1b341c29f19d3d9ac2fc05cc9111b37

Audit Summary

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

69	43	6	20
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	2
Medium	4
Low	5

Vulnerability	Severity
F-2024-1600 - Token Mismatch On Comission And Exchange	High
F-2024-1602 - Unverified `totalPrice` in Buyer Orders Leading to Commission Evasion	High
F-2024-1598 - Fee-on-Transfer Token Handling Flaw	Medium
F-2025-9635 - Executors Can Inflate Item Prices by Artificially Increasing Pool Size	Medium
F-2025-9638 - No Verification That Provided Funds Are Sufficient for All Reward Claims	Medium
F-2025-9639 - Executors Can Deposit NFTs on Behalf of Any User Who Has Approved the Contract	Medium
F-2024-1431 - Possible Overflows	Low
F-2024-1505 - Return values of `transfer()`/`transferFrom()` not checked	Low
F-2024-1598 - Do not allow fees to be set to `100%`	Low
F-2024-1640 - Lack of Restrictions in the `claim` Function	Low
F-2024-5425 - Checks-Effects-Interactions Pattern Violation	Low
F-2024-1444 - Missing checks for `address(0)`	Info
F-2024-1445 - Revert String Size Optimization	Info
F-2024-1446 - State Variables Only Set in The Constructor Should Be Declared `immutable`	Info
F-2024-1447 - Avoid Using State Variables Directly in `emit` for Gas Efficiency	Info
F-2024-1448 - Redundant import	Info
F-2024-1449 - Unneeded initializations of uint256 and bool variable to 0/false	Info

Vulnerability	Severity
F-2024-1462 - Public Functions That Should Be External	Info
F-2024-1464 - Convert duplicated codes to modifier/functions	Info
F-2024-1465 - Event is not properly `indexed`	Info
F-2024-1479 - Cache Variable Array Length In For Loop	Info
F-2024-1479 - Unsafe Usage of mint() and transferFrom()	Info
F-2024-1502 - Unnecessary Assert	Info
F-2024-1503 - Revert String Size Optimization	Info
F-2024-1504 - Unneeded initializations of uint256 and bool variable to 0/false	Info
F-2024-1547 - Missing Event Logging	Info
F-2024-1548 - Use `calldata` instead of `memory`	Info
F-2024-1574 - Cache Local Variable Array Length In For Loop	Info
F-2024-1575 - Revert String Size Optimization	Info
F-2024-1576 - Storage Layout Optimization	Info
F-2024-1577 - Unneeded initializations of uint256 and bool variable to 0/false	Info
F-2024-1578 - Public Functions That Should Be External	Info
F-2024-1580 - Style Guide Violation	Info
F-2024-1609 - Use `calldata` instead of `memory`	Info
F-2024-1641 - Missing checks for `address(0)`	Info
F-2024-1642 - Use `Ownable2Step` rather than `Ownable`	Info
F-2024-1643 - Owner Can Renounce Ownership	Info
F-2024-1644 - Cache Local Variable Array Length In For Loop	Info
F-2024-1646 - Revert String Size Optimization	Info
F-2024-1647 - Redundant import	Info
F-2024-1649 - Unneeded initializations of uint256 and bool variable to 0/false	Info
F-2024-1682 - Conditions Can Be Optimized	Info
F-2024-1684 - Unused Error Definition	Info
F-2024-1685 - Increments can be `unchecked` in for-loops	Info
F-2024-1705 - Missing Functionality	Info
F-2024-1706 - Add `unchecked {}` For Subtractions	Info
F-2024-1738 - Missing Event Logging	Info
F-2024-1739 - Unsafe Usage of transferFrom()	Info
F-2024-1803 - The `constructor` function lacks parameter validation	Info
F-2024-1804 - `Private` functions only called once can be inlined to save gas	Info
F-2024-1806 - Style guide: Function ordering does not follow the Solidity style guide	Info
F-2024-4573 - Cache Variable Array Length In For Loop	Info
F-2024-4575 - Public Functions That Should Be External	Info
F-2024-5426 - Missing Checks For Zero Address	Info
F-2024-5427 - Cache Variable Array Length In For Loop	Info

Vulnerability	Severity
F-2024-5428 - Conditions Can Be Optimized	Info
F-2024-5429 - Revert String Size Optimization	Info
F-2024-5431 - State Variables Only Set in The Constructor Should Be Declared as Immutable	Info
F-2024-5434 - Storage Layout Optimization	Info
F-2024-5436 - Revert String Size Optimization	Info
F-2024-5436 - Revert String Size Optimization	Info
F-2024-5443 - Potential Subtraction Operations Optimisation	Info
F-2024-5444 - Constants Declared as Public	Info
F-2024-5446 - Public Functions That Should Be External	Info
F-2024-5447 - Style Guide Violation	Info
F-2025-9634 - validAddress Modifier Prevents Using address(0x0) as Blackhole Address	Info
F-2025-9636 - Missing Sanity Check May Cause Unexpected Reverts with Panic Error	Info
F-2025-9637 - Incomplete Function Logic Leads to Discrepancy Between Implementation and Developer Intention	Info
F-2025-9640 - Deposited Equipment Items Are Not Moved to Item Pool After Round Ends, Contrary to Documentation	Info

Table of Contents

System Overview

MapleStory Universe is a virtual world ecosystem built on the MapleStory IP, featuring various games and applications, including the PC MMORPG "MapleStory N," that integrates blockchain technology and NFTs to enhance the gaming experience with the following contracts:

Exchange — provides essential functionality for validating signatures using EIP712 and SignatureChecker.

Exchange721 — extends Exchange to support trading of ERC-721 tokens. Uses structured orders for managing listings.

Exchange1155 — Extends Exchange to support trading of ERC-1155 tokens, including batch operations.

OrderMatch — handles order matching for both ERC-721 and ERC-1155 tokens, integrating with OrderBook.

OrderBook — manages the order book, facilitating order placement, matching, and cancellation. It extends OrderMatch and integrates with a commission system for creators.

Marketplace — the main marketplace contract, inherits from Exchange721, Exchange1155, and CommissionForCreator. It manages the marketplace operations including order matching and commission distribution.

MinProxyGetCode — provides a utility function to generate the bytecode for a minimal proxy contract that points to a specific implementation contract.

MinBeaconProxyGetCode — provides a utility function to generate the bytecode for a minimal beacon proxy contract that points to a specific beacon contract.

MinProxy — a library that leverages MinProxyGetCode and MinBeaconProxyGetCode to create minimal proxy and beacon proxy contracts. It encapsulates the logic to generate proxy bytecode and deploy the proxies.

ExecutorManager — manages the list of executors who have the privilege to generate or execute (cancel) transactions within the multisig contract.

ExecutorManagerUpgradeable — an upgradeable version of the ExecutorManager contract that manages executors within a multisig contract, ensuring compatibility with upgradeable contract patterns.

OwnerManager — manages the list of owners and the threshold for signing transactions within the multisig contract. Ensures that the required number of owner signatures is met before a transaction can be executed.

OwnerManagerUpgradeable — an upgradeable version of the OwnerManager contract that manages the owners and threshold within a multisig contract, ensuring compatibility with upgradeable contract patterns.

NextMesoToken — a ERC-20 wrapper of native NXPC token to be used within the Nexpace Limited dApps. Exchange of native NXPC to NESO at an immutable rate specified on deployment. The token parameters and features:

- Symbol: NESO
- Name: NextMeso
- Decimals: 18

Creator Wallet — a multisig wallet integrated with the ecosystem. The wallet enables supportof meta-transactions.

Bridge Entrypoint (Teller) — handles commission for bridge requests in NESO or NXPC tokens and forwards the execution to the actual bridge implementation. The contract requires the order to be signed by an off-chain service ensuring the commission amount is correct. The contract enables support of meta-transactions.

Commission — manages commission accrued by the creators allowing any time withdrawal. The contract enables support of meta-transactions.

DestinationBridge — manages cross-chain bridging of assets, including ERC-20, ERC-721, and ERC-1155 tokens.

ERC20BridgeToken — represents the ERC-20 tokens that are bridged and managed by the DestinationBridge.

ERC721BridgeToken — represents the ERC-721 tokens that are bridged and managed by the DestinationBridge.

ERC1155BridgeToken — represents the ERC-1155 tokens that are bridged and managed by the DestinationBridge.

Collection — facilitates management of player items by enabling the association of in-game assets with a player's wallet address.

MSNEquipSummary — stores the changing hash values of Equip items.

MSUContentsCommission —contract that is used to purchase off-chain goods such as MSN's Enhancement Contents by sending NextMeso (ERC20) to the Commission Wallet.

RandomSeedGenerator — uses the Chainlink VRF to generate the random numbers needed to enhance an item.

ERC20ApproveControlled — an extension of the ERC20 token standard. The contract introduces functionality where only approved operators can be given infinite allowances if they are also approved by the token owner.

ERC721ApproveControlled — an extension of the ERC721 token standard and ApproveControlled contracts, that provides an extra layer of approval control for NFT s, where only approved operators can be granted approval for all tokens of a particular owner if they are also approved by the token owner.

ERC1155ApproveControlled — The contract extends the ERC1155 token standard and ApproveControlled contracts to provide a controlled approval system for multi-token

management, wherein approval for all token types of a specific owner can only be given to an approved operator if they are also approved by the token owner.

NextOwnable — The contract provides a flexible permission system where critical owner-level operations can be limited, and operations can be managed by trusted executors.

NextOwnablePausable — The contract builds on the NextOwnable contract by integrating the Pausable functionality from OpenZeppelin, allowing the contract to be paused and unpaused by the owner or executor

ApproveControlled — The contract provides functionalities for the owner to approve or disapprove operators. This approval mechanism enables a selective control over function execution.

ApproveController — The contract provides mechanisms for an account to set its approval status, and for the owner to allowlist an account.

Exec — The contract provides external functions called exec, batchExec to allow the contract to cause arbitrary transactions to occur, such as in a smart contract wallet. For extensibility, virtual functions are defined that act as callback functions.

ExecNonPayable — Non-payable version of Exec contract.

ERC2771ContextConstant — The contract extends the Context contract and modifies it to handle meta-transactions through the use of a trusted forwarder.

BaseFactoryBeacon — an extension of the UpgradeableBeacon contract from the OpenZeppelin library.

FactoryBeacon — provides a method to deploy a new proxy contract and then makes a function call to the newly deployed contract in one transaction.

SaltyFactoryBeacon — The contract deploys a new proxy contract using the Create2.deploy function, with the provided salt.

SupervisedOwnable — The contract is an abstract contract that introduces a layer of access control. This contract defines a supervisor, a role that can perform certain restricted operations.

StorageSupervisedOwnable — The contract extends from SupervisedOwnable and provides a concrete implementation for the supervisor function. It provides a way to set and get the supervisor contract.

NextOwnablePausableSupervisor — The contract provides view functions for the supervisor.

ImmutableSupervisedOwnable — Immutable version of SupervisedOwnable contract.

NXPCDistributor — NXPC distributor contract for equipment item deposits and withdrawals

ItemIssuance — A contract to issue Nexpace items

Privileged roles

Owner Role

- The Owner has the highest level of access and control within the contract. This role is responsible for managing critical functions such as asset management, contract pausing, and ownership management.
  - Can perform asset-related tasks like clearing slot information or withdrawing ERC-20/ERC-721 tokens.
  - Can update or set key configurations, such as the key hash for the `RandomSeedGenerator`.
  - Has the authority to manage blocklists, checkpoint addresses, add relayers, and update bridge contracts.

Executor Role

- The Executor is an administrator role responsible for overseeing specific in-game and contract-related activities. This role is typically focused on managing user items, token deposits, and performing operations on behalf of users.
  - Manages in-game item collections and enhances character stats.
  - Has permission to deposit the `NextMeso` ERC-20 token into the commission wallet.
  - Can pause transfer and exchange functionalities in the `NextMeso` token.
  - Executes various contract-related operations, including managing minimum issue amounts and adding/removing items.
  - Executor roles can update critical contract settings such as commission tokens and can pause certain operations in the bridge contracts.

CommissionForCreator

- Responsible for managing and distributing commissions to creators within the system.

SelfCall

- All management functions such as adding/removing owners and executors, and changing thresholds, must be executed through a multisig transaction process, ensuring that these changes are made securely.

Potential Risks

Scope Limitation: This assessment was conducted on separate occasions by different teams and auditors, and each contract was audited individually at different times. As such, the overall flow of the system, including how the contracts interact with one another, was not tested as a whole.

Contract Interaction Assumption: This audit report focuses exclusively on the security evaluation of the individual contracts within the review scope. Interactions between these contracts, or with any external contracts not covered by this audit, are assumed to be correct but were not tested or verified as part of this assessment. It is important to note that since each contract was audited separately, the potential risks arising from their combined use or integration have not been fully explored.
Given that this audit does not account for the interactions across the entire system, the overall security of the ecosystem, including external contracts and integrations beyond the reviewed scope, cannot be guaranteed.

Single Points of Failure and Control: If the executor becomes compromised, unavailable, or behaves maliciously, it could disrupt core functionalities, block user access to rewards, or manipulate the protocol in unintended ways. In addition, all equipment items are transferred into Vault. In case the Vault address is compromised, all items can be stolen.

Flexibility and Risk in Contract Upgrades: The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.

Lack of Transparency & Auditability: Decisions made by the executor (e.g., how rewards are calculated or who gets included in a Merkle tree) may not be easily verifiable by external observers, reducing the system’s trustworthiness.

External Calls in Iteration Risks: Making external calls within loops increases the risk of gas exhaustion, potentially leading to failed transactions and reduced contract reliability, especially when processing large datasets.

Contract Specific Risks

Marketplace:

- The functionality for item sales is managed off-chain, with no corresponding mechanisms implemented on-chain.
- The contract exhibits a high degree of centralization, with the owner and/or executor having exclusive control over nearly all functions. This includes the atomicmatch function, which facilitates token swaps, being restricted to execution by the executor alone.

Nexpace Contracts:

- The contract exhibits a high degree of centralization, with the owner and/or executor having exclusive control over nearly all functions. This includes the claim function, which helps creator to claim the fees.

NFT:

- The owner role can clear the slots using the `clearSlot()` function for any wallet address at any time. In this case, the user will not be able to retrieve the NFT, and it will remain held within the contract.
- The owner role can withdraw any user's NFT's held in the contract at any time.
- The functionalities are highly centralized, leaving regular users almost unable to call anything except for getter functions. Nearly every function is restricted to be called by the owner or executor only.

Maplestory:

- The contract contains several `onlyOwner` functions, such as `retrieveERC1155` and `retrieveERC721`, that grant the contract owner the ability to transfer any ERC1155 or ERC721 tokens from any user to any address. This centralization of control poses a significant risk, as it allows the owner to unilaterally move users' tokens without their consent once the contracts have been approved by the users. Such a design concentrates power and trust in the hands of the contract owner, potentially exposing users to malicious actions or abuse of power.

Findings

Vulnerability Details

F-2024-1600 - Token Mismatch On Comission And Exchange - High

Description:

The `atomicMatch1155` function within a smart contract designed for facilitating transactions between buyers and sellers of ERC-1155 tokens identifies a vulnerability associated with commission calculation and token exchange mechanism. The issue arises due to the lack of a stringent check ensuring that the token used for commission payments (`_token`) is the same as the token involved in the buyer and seller orders (`sellerOrder.tokenAddress`). Given the variance in decimal places among ERC-20 tokens (e.g., USDT vs. USDC), the lack of this check allows for potential manipulation where commissions paid could be significantly lower than expected, undermining the integrity of the marketplace's fee structure.

```
/// @notice Get the token address
function token() external view returns (IERC20) {
    return _token;
}

/// @notice Send the commission
function _sendCommission(CommissionParams memory params) internal {
    if (params.commissionAmount != 0) {
        _token.safeTransferFrom(params.commissionFrom, _commission, params.co
mmissionAmount);
        Commission(_commission).afterDeposited(params.commissionTo, params.co
mmissionAmount);
        emit SendCommission(
            params.commissionFrom,
            params.commissionTo,
            params.commissionAmount,
            params.dAppId,
            params.reason
        );
    }
}
```

```
function _tokenExchange(
    SellerOrder1155 memory sellerOrder,
```

```
    BuyerOrder1155 memory buyerOrder,
    uint256 index,
    uint256 actualPrice
) internal {
    IERC20(sellerOrder.tokenAddress).safeTransferFrom(
        buyerOrder.buyerAddress,
        sellerOrder.sellerAddress,
        actualPrice
    );
    IERC1155(sellerOrder.ftAddress).safeTransferFrom(
        sellerOrder.sellerAddress,
        buyerOrder.buyerAddress,
        sellerOrder.ftTokenId,
        buyerOrder.amounts[index],
        ""
    );
}
```

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Medium

Severity: High

Recommendations

Remediation:

```
function atomicMatch1155(
    // Parameters
) external whenExecutable {
    // Existing checks
    require(commission.commissionTo != address(0), "Invalid commission recipient");
    require(commission.commissionPercentage <= 10000, "Invalid commission percentage");
```

```
    // Check for token consistency
    // this token should be same token under comission.
    require(_token == buyerOrder.tokenAddress, "Mismatch in transaction and c
ommission tokens");

    // Rest of the function implementation
}
```

Remediation (Revised commit: 291d841): The Nexon team resolved the issue by implementing the same token address for token transfer and the commission contract.

F-2024-1602 - Unverified `totalPrice` in Buyer Orders Leading to Commission Evasion - High

Description: The smart contract mechanism responsible for handling transactions between buyers and sellers of ERC-1155 tokens contains a oversight in the validation of `totalPrice` provided by the buyer within the `atomicMatch1155` function. The identified vulnerability stems from the contract's failure to ensure that the `totalPrice` specified by the buyer corresponds accurately to the cumulative price of the seller's orders, considering the quantities and prices of assets being transacted. This lapse in validation could potentially be exploited by a malicious actor to input a significantly reduced `totalPrice`, including zero, effectively circumventing the intended commission fees and undermining the marketplace's revenue generation mechanism.

```
function atomicMatch721(
    Order721 memory sellerOrder,
    Order721 memory buyerOrder,
    bytes[2] calldata signatures,
    Commission memory commission
) external whenExecutable {
    require(commission.commissionTo != address(0), "Marketplace/invalidReques
t: wrong commission to address");
    require(commission.commissionPercentage <= 10000, "Marketplace/invalidReq
uest: wrong commission percentage");
    _atomicMatch721(sellerOrder, buyerOrder, signatures);

    // Commission
    uint256 commissionAmount = ((buyerOrder.tokenAmount * commission.commissi
onPercentage) / 10000);
```

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Mitigated

Classification

Impact: 4/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation: To ensure accurate commission calculation and prevent potential exploitation, the following validation step can be integrated into the `atomicMatch1155` function:

```
function atomicMatch1155(
    // Parameters
) external whenExecutable {
    uint256 calculatedTotalPrice = 0;
    // Loop through seller orders to calculate the expected total price
    for (uint256 i = 0; i < sellerOrders.length; i++) {
        calculatedTotalPrice += sellerOrders[i].tokenAmount * buyerOrder.amou
nts[i];
    }
    // Verify the calculatedTotalPrice against the buyer's declared totalPric
e
    require(calculatedTotalPrice == buyerOrder.totalPrice, "Invalid totalPric
e: does not match seller orders");

    // Proceed with commission calculation and the rest of the function
}
```

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> This validation proved to consume a significant amount of gas, so ultimately, we decided to perform this validation off-chain.

F-2024-1598 - Fee-on-Transfer Token Handling Flaw - Medium

Description:

In ERC20 token exchanges, the contract assumes that the amount of tokens sent is the amount received by the other party. This assumption holds true for standard ERC20 tokens but fails for fee-on-transfer (FoT) tokens, which automatically deduct a transaction fee from the transfer amount, reducing the final amount received. Specifically, the Exchange contracts performs token swaps via `_atomicMatch721()` and `_atomicMatch1155()` without verifying that the received amount matches the expected amount after accounting for potential transfer fees.

```
    function _atomicMatch721(
        Order721 memory sellerOrder,
        Order721 memory buyerOrder,
        bytes[2] memory signatures
    ) internal {
       …
        _validateSignature(sellerHash, sellerOrder.maker, signatures[0]);
        _validateSignature(buyerHash, buyerOrder.maker, signatures[1]);
        /* Checks effects interactions pattern*/
        _fulfill[sellerHash] = true;
        _fulfill[buyerHash] = true;
        /* Actually send stuffs */
        /* Seller -(NFT)-> buyer. */
        IERC721(sellerOrder.nftAddress).transferFrom(sellerOrder.maker, buyerO
rder.maker, sellerOrder.nftTokenId);
        /* Buyer -(ERC20 Token)-> seller. */
        IERC20(sellerOrder.tokenAddress).safeTransferFrom(buyerOrder.maker, se
llerOrder.maker, buyerOrder.tokenAmount);
        /* Log match event. */
        emit Exchange721Matched(sellerHash, buyerHash, sellerOrder.maker, buye
rOrder.maker, buyerOrder.tokenAmount);
    }
```

Similarly, the Marketplace contract calculates commission based on the initial token amount specified for the transaction, disregarding the actual amount received after fee-on-transfer (FoT) deductions,.

```
    function atomicMatch1155(
        SellerOrder1155[] memory sellerOrders,
```

```
        BuyerOrder1155 memory buyerOrder,
        bytes[] calldata sellerSignatures,
        bytes calldata buyerSignature,
        Commission memory commission
    ) external whenExecutable {
        …
        uint256 commissionAmount = (buyerOrder.totalPrice * commission.commiss
ionPercentage) / 10000;
        _sendCommission(
            CommissionForCreator.CommissionParams({
                commissionFrom: buyerOrder.buyerAddress,
                commissionTo: commission.commissionTo,
                dAppId: commission.dAppId,
                commissionAmount: commissionAmount,
                reason: "MarketPlace: atomicMatch1155"
            })
        );
        …
    }
```

This can result in the recipient receiving less than the initially specified amount.

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Mitigated

Classification

Impact: 3/5

Likelihood: 4/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Utilize the balanceOf function to check the reciever's token balance before and after executing the transferFrom call. This method allows for the precise calculation of the actual amount of tokens received, accounting for any fees deducted during the transfer.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> 'FoT' tokens are irrelevant to us. We deploy our own Marketplace contract on a subnet and even control smart contract deployment.

F-2025-9635 - Executors Can Inflate Item Prices by Artificially Increasing Pool Size - Medium

Description:

The `addItem()` function allows authorized executor addresses to increase the pool amount of a specific universe without any upper bound. Since item pricing relies on the `poolAmount` via the `expectAmount()` function (which calculates price based on `totalSupply / poolAmount`), a malicious executor can artificially inflate item prices by repeatedly calling `addItem()` with large values.

This mechanism poses a significant risk as executors are trusted middleware, and their misuse—intentional or accidental—can disrupt pricing logic and negatively impact users interacting with the protocol. Additionally, the lack of an upper bound on the `amount` parameter further exacerbates this issue, enabling unbounded manipulation.

ItemIssuance.sol:

```
function addItem(uint256 universe, uint256 amount) external whenUniverseExist
s(universe) whenExecutable {
    universes[universe - 1].poolAmount += amount;

    emit ItemAdded(universe, amount);
}
```

ItemIssuance.sol:

```
function expectAmount(uint256 universe, uint256 amount) public view whenUnive
rseExists(universe) returns (uint256) {
    uint256 itemAmount = universes[universe - 1].poolAmount;

    require(amount <= itemAmount, "ItemIssuance/invalidAmount: too large amou
nt");

    return Math.ceilDiv(NXPCAmountManager.totalSupply() * amount, itemAmount)
;
}
```

Assets:

- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Accepted

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Limit the maximum value that can be passed to the `amount` parameter in the `addItem()` function to avoid excessive pool inflation. In addition, maintain logs or metrics per executor and flag unusual behavior, such as sudden spikes in added amounts.

Resolution: This finding was **acknowledged** by the client with a notice:

> Conceptually, the item quantity periodically moves from the ItemBase to the ItemPool. When the quantity in the ItemPool increases, the item price decreases.
> The cycle progression and the amount transferred are entirely handled off-chain. Since validation is done off-chain, I believe on-chain validation is not necessary.

F-2025-9638 - No Verification That Provided Funds Are Sufficient for All Reward Claims - Medium

Description:

The `setReward()` function allows the executor to register a Merkle root and send NXPC to fund reward claims for a given round. However, **there is no verification** that the total sum of rewards embedded in the Merkle tree actually **matches or is less than** the NXPC provided (`msg.value`).

This oversight introduces a serious **availability and fairness risk**: if the Merkle tree promises more rewards than the contract holds, only the first claimants will successfully withdraw funds. Remaining users—despite being correctly included in the tree and fulfilling participation requirements—will receive nothing.

This situation may lead to some depositors in the protocol not receiving their rewards due to inadequate rewards.

NXPCDistributor.sol:

```
function setReward(uint256 round, bytes32 merkleRoot) external payable whenExecutable {
    require(currentRound() > round, "NXPCDistributor/invalidRound: round must be ended");

    NXPCAmountManager.addMintedAmount(msg.value);

    _setReward(round, merkleRoot);
}
```

NXPCDistributor.sol:

```
function claim(uint256 round, address user, uint256 amount, bytes32[] calldata proof) external whenExecutable {
    require(isClaimable(round), "NXPCDistributor/notClaimable: reward has not been registered");
    require(
        MerkleProof.verifyCalldata(
            proof,
            rewardMerkleRoot(round),
            keccak256(bytes.concat(keccak256(abi.encode(round, user, amount))))
        ),
        "NXPCDistributor/invalidProof: reward merkle root is different"
    );
```

```
    _claim(round, user, amount);
}
```

NXPCDistributor.sol:

```
function _claim(uint256 round, address user, uint256 amount) private {
    bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(round, user, amount))));

    require(!isClaimed(leaf), "NXPCDistributor/alreadyClaimed: merkle leaf is already used");

    _isClaimed[leaf] = true;

    //slither-disable-next-line arbitrary-send-eth
    (bool success, ) = user.call{ value: amount }("");

    require(success, "NXPCDistributor/transferFailed: NXPC transfer failed");

    emit Claim(round, amount, user);
}
```

Assets:

- NXPCDistributor.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Mitigated

Classification

Impact: 4/5

Likelihood: 3/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Store the total expected reward amount per round on-chain to allow public verification and reduce reliance on off-chain assumptions. Then, modify `setReward()` to require an additional parameter: the total amount encoded in the Merkle tree, and enforce that `msg.value >= totalAmount`.

Resolution: This finding is **mitigated** by client's statement:

> We use a Merkle root to avoid full validation and gain gas efficiency.
> In practice, only our executor can register rewards, and all value transfers for generating the Merkle root and registering rewards are handled off-chain.
> If a reward is registered incorrectly, we can simply call `setReward` again to correct it.

F-2025-9639 - Executors Can Deposit NFTs on Behalf of Any User Who Has Approved the Contract - Medium

Description:

The `deposit()` function allows an authorized executor to transfer NFTs into the system on behalf of any user, provided that the user has approved the contract. Also, the function accepts a `user` address as an argument, but this value is **not used** as part of the Merkle tree leaf—meaning it is not cryptographically verified during proof validation.

As a result, a **malicious executor** could:

- Scan for users who have approved the contract,
- Construct a Merkle tree containing only eligible `itemId` and `slotLength` values (without tying them to a specific `user`),
- Call `deposit()` to transfer NFTs owned by any approved user into the vault without their knowledge.

This creates a serious security concern, as it undermines user consent and exposes approved NFTs to unauthorized actions. According to normal workflow, users should call the deposit function over a legitimate **executor** on their own. Considering that the executor address here works as middleware, a compromise of this role can lead to unintended deposits of NFTs.

NXPCDistributor.sol:

```
function deposit(
    uint256 tokenId,
    address user,
    uint256 slotLength,
    bytes32[] calldata proof
) external whenExecutable validAddress(user) {
    require(isStarted(), "NXPCDistributor/alreadyEnded: round must be started
");

    uint64 itemId = equip.tokenItemId(tokenId);

    require(
        MerkleProof.verifyCalldata(
            proof,
            basketMerkleRoot(currentRound()),
            keccak256(bytes.concat(keccak256(abi.encode(currentRound(), itemI
d, slotLength))))
        ),
        "NXPCDistributor/invalidProof: basket merkle root is different"
    );
```

```
    _deposit(currentRound(), tokenId, user, slotLength);
}
```

Assets:

- NXPCDistributor.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Mitigated

Classification

Impact: 4/5

Likelihood: 3/5

Exploitability: Semi-Dependent

Complexity: Medium

Severity: Medium

Recommendations

Remediation: Modify the Merkle leaf encoding to include the `user` address. For example:

```
keccak256(abi.encode(currentRound(), user, itemId, slotLength))
```

This ensures the Merkle tree only authorizes specific users for deposits and prevents generalized misuse.

Resolution: The finding is **mitigated** with the client's statement regarding this design:

> The Merkle root does not guarantee that a specific user can deposit a specific item.
> It simply indicates which items can be deposited, and the users who can deposit them are arbitrary.
> Therefore, user addresses cannot be included in the Merkle root.

F-2024-1431 - Possible Overflows - Low

Description:

The `setOffChainUsedAmount()` function within the `MaplestoryConsume` contract is designed to update `offChainUsedAmount` and `offChainAmount` for specified token IDs based on the input amounts. While the function includes a check to ensure that the input amounts do not exceed **(2**64/2)**, it lacks validation against the cumulative effect of multiple executions on `token.offChainAmount` and `token.offChainUsedAmount`. This oversight can lead to integer overflow conditions

- **offChainUsedAmount Overflow :** The function allows `token.offChainUsedAmount` to be incremented without bounds checking beyond the initial input validation. If an executor calls `setOffChainUsedAmount` twice with **((2**64/2) - 1)** amount for the same token ID that cumulatively exceed the maximum **uint64** value (**2**64 - 1**), it can cause an overflow in `token.offChainUsedAmount.`
- **offChainAmount Overflow :** Similarly, `token.offChainAmount` is decremented without validating the operation against the current state. Repeated calls can lead to an underflow if the cumulative deducted amount exceeds the token's `offChainAmount.` For example if an executor call `setOffChainUsedAmount` thrice with **((2**64/2) - 1)** amount for the same token ID that cumulatively exceed the **int64** value and it can cause an underflow in `token.offChainAmount`

Additionally `setIssuedAmount()` function is also designed to increase token.issuedAmount and `token.offChainAmount` amounts similar overflow scenarios can also occur in the `setIssuedAmount()`

Overflow conditions can corrupt the token accounting logic, leading to inaccurate tracking of off-chain used and issued amounts.

```
    /// @notice to convert token into game consumable. consumed in-game, and
reported by setOffChainUsed.
    function melt(address from, uint256 id, uint256 amount) external whenExec
utable {
        require(amount < 9223372036854775808, "MaplestoryConsume/amountOverfl
ow: amount's max value is int64");
        Token storage token = _tokens[id];
        _burn(from, id, amount);
        unchecked {
            token.onChainAmount -= uint64(amount);
            token.offChainAmount += int64(uint64(amount));
        }
    }
```

```
    /// @notice minting. _mint function is from ERC1155.
    /// you want to mint id token by amount.
    /// commission.reason means that this commission comes from "MaplestoryCo
nsume: mint" function
    function mint(address to, uint256 id, uint256 amount, bytes calldata data
) external whenExecutable {
        require(amount < 9223372036854775808, "MaplestoryConsume/amountOverfl
ow: amount's max value is int64");
        Token storage token = _tokens[id];
        unchecked {
            token.onChainAmount += uint64(amount);
            token.offChainAmount -= int64(uint64(amount));
        }
        _mint(to, id, amount, data);
    }
```

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Introduce checks to validate the cumulative effect of updates on `token.offChainUsedAmount` , `token.issuedAmount` and `token.offChainAmount`. Before applying updates, calculate the new total and verify that it does not exceed the bounds of **uint64** and **int64.**

Remediation (Revised commit: 83ccf0b): The Nexon team fixed the issue by removing `unchecked` keywords.

F-2024-1505 - Return values of `transfer()`/`transferFrom()` not checked - Low

Description: Not all `ERC20` implementations `revert()` when there's a failure in `transfer()` or `transferFrom()`. The function signature has a boolean return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually transfer anything.

```
token.transfer(tos[i], amounts[i]);
```

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Fixed

Classification

Impact: 2/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: To ensure the reliability and security of token transfers in your smart contract, it's crucial to check the return values of the `transfer()` and `transferFrom()` functions. These functions often return a boolean value indicating the success or failure of the transfer operation. By checking this return value, you can accurately determine whether the transfer was successful and handle any potential errors or failures accordingly. Failing to check the return value may lead to unintended and unhandled transfer failures, which could have security and usability implications.

Remediation (Revised commit: e4306ee): The Nexon team fixed the issue by implementing the `safeTransfer` instead of `transfer`.

F-2024-1598 - Do not allow fees to be set to `100%` - Low

Description: It is recommended from a risk perspective to disallow setting 100% fees at all. A reasonable fee maximum should be checked for instead.

```
function atomicMatch1155(
    SellerOrder1155[] memory sellerOrders,
    BuyerOrder1155 memory buyerOrder,
    bytes[] calldata sellerSignatures,
    bytes calldata buyerSignature,
    Commission memory commission
) external whenExecutable {
    require(commission.commissionTo != address(0), "Marketplace/invalidReques
t: wrong commission to address");
    require(commission.commissionPercentage <= 10000, "Marketplace/invalidReq
uest: wrong commission percentage");

    // Commission
    uint256 commissionAmount = (buyerOrder.totalPrice * commission.commission
Percentage) / 10000;
```

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Mitigated

Classification

Impact: 2/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Implement strict validation checks in your smart contract's fee-setting functions to ensure that fees cannot be configured to 100%. Define a maximum allowable fee percentage that preserves the

contract's functionality and user experience, such as 10% or 20%, depending on your project's specific requirements and economic model. For example:

```
function setTransactionFee(uint256 newFeePercentage) public onlyOwner {
    require(newFeePercentage <= 20, "Fee cannot exceed 20%");
    transactionFee = newFeePercentage;
}
```

Remediation (Mitigated): The function is restricted, but the fee percentage can still reach up to 99%; the Nexon team has accepted the risk.

F-2024-1640 - Lack of Restrictions in the `claim` Function - Low

Description:

The Solidity contract function `claim(address creator, uint256 claimAmount, uint256 burnAmount)` is intended to allow executors to claim fees on behalf of a creator by transferring a specified `claimAmount` to the creator's address and burning a `burnAmount` of the token. This function is critical for managing the distribution and burning of tokens within a decentralized application.

A significant oversight in the design of this function is the absence of any checks or restrictions on the proportions of `claimAmount` and `burnAmount` relative to the total amount available to the creator (`_creatorFees[creator]`). Specifically, there is no safeguard to prevent the executor from setting the `burnAmount` to a value that effectively nullifies the `claimAmount`, potentially burning 100% of the tokens intended for the creator. This issue exposes a vulnerability where an executor could, whether by error or malfeasance, burn all of a creator's tokens without transferring any to the intended recipient.

```
function claim(address creator, uint256 claimAmount, uint256 burnAmount) exte
rnal onlyAtLeastExecutor {
    _creatorFees[creator] -= claimAmount + burnAmount;
    _latestTokenAmount -= claimAmount + burnAmount;

    _token.safeTransfer(creator, claimAmount);
    ERC20Burnable(address(_token)).burn(burnAmount);

    emit Claimed(creator, claimAmount, burnAmount, _creatorFees[creator]);
}
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Mitigated

Classification

Impact: 2/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: To address this vulnerability and ensure the integrity of token distribution and burning, the following recommendation is proposed:

```
function claim(address creator, uint256 claimAmount, uint256 burnAmount) exte
rnal onlyAtLeastExecutor {
    require(_creatorFees[creator] >= claimAmount + burnAmount, "Insufficient
funds to claim and burn");
    require(burnAmount <= _creatorFees[creator] * maxBurnPercentage / 100, "B
urn amount exceeds limit");

    _creatorFees[creator] -= claimAmount + burnAmount;
    _latestTokenAmount -= claimAmount + burnAmount;

    _token.safeTransfer(creator, claimAmount);
    ERC20Burnable(address(_token)).burn(burnAmount);

    emit Claimed(creator, claimAmount, burnAmount, _creatorFees[creator]);
}
```

In this example, `maxBurnPercentage` is a predefined constant that caps the `burnAmount` to a sensible fraction of the creator's total fees, thereby preventing total depletion through burning.

Remediation (Mitigated): The `require(_creatorFees[creator] >= claimAmount + burnAmount, "Commission: insufficient funds to claim and burn");` check was implemented, and The Nexon team mitigated the issue with the following statement;

> We can burn all amount of creator balance in commission contract and also burnAmount can be zero.

F-2024-5425 - Checks-Effects-Interactions Pattern Violation - Low

Description: State variables are updated after the external calls to the token contract.

As explained in Solidity Security Considerations, it is best practice to follow the checks-effects-interactions pattern when interacting with external contracts to avoid reentrancy-related issues.

The balance should be set before calling the `.call` function, following code breaks CEI pattern:

```
(bool success, ) = recipient.call{ value: amount }("");
require(success, "SourceBridge/transferFailed: Transfer failed");
bridgedBalances[sourceBlockchainID][sourceBridgeAddress][sourceTokenAddress]
= currentBalance - amount;
```

Assets:

- SourceBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Follow the checks-effects-interactions pattern when interacting with external contracts.

Remediation The Nexon team fixed the issue in the commit: `6b1b3ea` with following the CEI pattern.

F-2024-1444 - Missing checks for `address(0)` - Info

Description: In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. The "
Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.
For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

```
_character = character;
```

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Mitigated

Classification

Impact: 1/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: It is strongly recommended to implement checks to prevent the zero address from being set during the initialization of contracts. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

The mentioned assets process did not include a verification step because contracts directly created by the smart contract cannot have the address(0) value.

F-2024-1445 - Revert String Size Optimization - Info

Description:

Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore` , along with additional overhead to calculate memory offset, etc.

Number Of Instances

There are 27 instances of this issue

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation:

To optimize Gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> Our functions are mostly called through our gRPC. Since we use gRPC for communication, validations are performed off-chain. Consequently, there are very few calls that would revert. We believe that this aspect doesn't need to be applied.

F-2024-1446 - State Variables Only Set in The Constructor Should Be Declared `immutable` - Info

Description: Compared to regular state variables, the gas costs of constant and immutable variables are much lower. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed.

```
ItemIssuance private _itemIssuance;
```

This variable can be declared immutable. This will lower the Gas taxes.

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Consider marking state variables as an immutable that never changes on the contract.

Remediation (Revised commit: 1f7c4f9): The Nexon team fixed the issue by adding `immutable` keyword.

F-2024-1447 - Avoid Using State Variables Directly in `emit` for Gas Efficiency - Info

Description: In Solidity, emitting events is a common way to log contract activity and changes, especially for off-chain monitoring and interfacing. However, using state variables directly in `emit` statements can lead to increased gas costs. Each access to a state variable incurs gas due to storage reading operations. When these variables are used directly in `emit` statements, especially within functions that perform multiple operations, the cumulative gas cost can become significant. Instead, caching state variables in memory and using these local copies in `emit` statements can optimize gas usage.

```
emit CycleUpdated(lastUpdateTime, period, amountPerPeriod);
emit MintableAmountAdded(uint256(periodsPassed) * amountPerPeriod, mintableAm
ount);
```

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To optimize gas efficiency, cache state variables in memory when they are used multiple times within a function, including in `emit` statements.

Remediation (Revised commit: 45e6e56): The Nexon team fixed the issue by caching the state variable.

F-2024-1448 - Redundant import - Info

Description:

These contracts

```
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { IERC721Metadata } from "@openzeppelin/contracts/token/ERC721/extensi
ons/IERC721Metadata.sol";
```

are imported in files `MaplestoryConsume.sol`, `MaplestoryEquip.sol` and `MaplestoryCharacter.sol` but never used.

This redundancy in import operations has the potential to result in unnecessary gas consumption during deployment and could potentially impact the overall code quality.

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

Remove redundant imports, and ensure that the contract is imported only in the required locations, avoiding unnecessary duplications.

Remediation (Revised commit: 0a2e5d7): The Nexon team fixed the issue by removing redundant imports.

F-2024-1449 - Unneeded initializations of uint256 and bool variable to 0/false - Info

Description:

In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing `uint256` variables to `0` and `bool` variables to `false` when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

Number Of Instances

There are 17 instances of this issue

```
for (uint256 i = 0; ...
```

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

It is recommended not to initialize integer variables to `0` to and boolean variables to `false` to save some Gas.

Remediation (Revised commit: 185f947): The Nexon team fixed the issue by removing unneeded initializations.

F-2024-1462 - Public Functions That Should Be External - Info

Description: Functions that are meant to be exclusively invoked from external sources should be designated as `external` rather than `public`. This is essential to enhance both the gas efficiency and the overall security of the contract.

```
function balanceOfBatch(address[] memory accounts) public view returns (uint256[] memory) {
...
function predictMintableAmount() public view returns (uint256) {
```

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To optimize gas usage and improve code clarity, declare functions that are not called internally within the contract and are intended for external access as `external` rather than `public`. This ensures that these functions are only callable externally, reducing unnecessary gas consumption and potential security risks.

Remediation (Revised commit: 14e106e): The Nexon team fixed the issue by changing visibility from `public` to `external`.

F-2024-1464 - Convert duplicated codes to modifier/functions - Info

Description:

Duplicated code in Solidity contracts, especially when repeated across multiple functions, can lead to inefficiencies and increased maintenance challenges. It not only bloats the contract but also makes it harder to implement changes consistently across the codebase. By identifying common patterns or checks that are repeated and abstracting them into modifiers or separate functions, the code can be made more concise, readable, and maintainable. This practice not only reduces the overall bytecode size, potentially lowering deployment and execution costs, but also enhances the contract's reliability by ensuring consistency in how these common checks or operations are executed.

```
require(amount < 9223372036854775808, "MaplestoryConsume/amountOverflow: amou
nt's max value is int64");
Token storage token = _tokens[id];
_burn(from, id, amount);
```

This 3 lines of code used in both `burn` and `melt` functions.

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

Identify and consolidate duplicated code blocks in Solidity contracts into reusable modifiers or functions. This approach streamlines the contract by eliminating redundancy, thereby improving clarity, maintainability, and potentially reducing gas costs. For common checks, consider using modifiers for concise and consistent enforcement of conditions. For reusable logic, encapsulate it in functions to avoid code duplication and simplify future updates or bug fixes.

Remediation (Revised commit: e1b12a0): The Nexon team fixed the issue by converting duplicated require statements to modifiers..

F-2024-1465 - Event is not properly `indexed` - Info

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. This is especially useful when it comes to filtering based on an address. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Where applicable, each `event` should use three `indexed` fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three applicable fields, all of the applicable fields should be `indexed`.

```
event RetrievedNeso(address from, address to, uint256 amount, string reason);
...
event RetrievedERC1155(address from, address to, uint256 id, uint256 amount,
string reason);
...
event RetrievedERC721(address from, address to, uint256 tokenId, string reaso
n);
```

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation: Enhance smart contract efficiency post-deployment by utilizing indexed events. This approach aids in efficiently tracking contract activities, significantly contributing to the reduction of gas costs.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> We operate a crawler to track all transactions in the off-chain area. Since this event occurs too frequently, we believe that using tracking through the crawler is preferable to adding indexed fields.

F-2024-1479 - Cache Variable Array Length In For Loop - Info

Description: If not cached, the solidity compiler will always read the length of the array during each iteration. If it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

There are 10 instances of this issue:

```
for (uint256 i = 0; i < accounts.length; ++i)
for (uint256 i = 0; i < mints.length; )
for (uint256 i = 0; i < tokenIds.length; )
for (uint256 i = 0; i < tokenIds.length; ++i)
for (uint256 i = 0; i < characterIds.length; ++i)
```

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To optimize gas consumption, cache the length of memory arrays before for loop iterations in Solidity. This reduces redundant mload operations, saving 3 gas per iteration after the first and improving overall contract efficiency.

Remediation (Revised commit: 44f1162): The Nexon team fixed the issue by caching array lengths.

F-2024-1479 - Unsafe Usage of mint() and transferFrom() - Info

Description: The smart contracts in question utilizes the `ERC721._mint()` method for minting new Non-Fungible Tokens (NFTs) and `transferFrom()` for transferring NFTs between addresses. While these methods are functionally correct, they lack the safety mechanisms provided by their counterparts, `_safeMint()` and `safeTransferFrom()`, especially in scenarios where the recipient is a smart contract. The absence of such safety checks can result in NFTs being permanently locked or lost if the recipient contract does not implement the ERC721 token receiver interface (IERC721Receiver).

Assets:

- MaplestoryCharacter.sol [https://github.com/nexpace-dev/maple-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

- **Use _safeMint() for Minting:** Replace all instances of **ERC721._mint()** with **ERC721._safeMint()** to ensure that the recipient contract can handle the minted NFT. This change adds a crucial safety check for contracts receiving NFTs.
- **Use safeTransferFrom() for Transfers:** Replace **transferFrom()** with **safeTransferFrom()** for all NFT transfers. This ensures that the receiving contract is capable of handling ERC721 tokens, preventing accidental loss.
- **Implement Checks-Effects-Interactions Pattern:** When integrating the recommended changes, ensure that the contract's functions adhere to the checks-effects-interactions pattern to prevent potential reentrancy attacks. This involves performing all necessary conditions and state changes before interacting with external contracts.

Remediation (Revised commit: b47c32c): The Nexon team fixed the issue by adding safe versions of those functions and implementing the checks-effects-interaction pattern.

F-2024-1502 - Unnecessary Assert - Info

Description: Unnecessary `assert` statements in Solidity contracts can have several detrimental impacts on code readability, gas efficiency, and overall contract functionality. `assert` statements are typically used to check for conditions that should never occur under normal circumstances. While they can be valuable for catching unexpected errors and halting contract execution, their misuse can lead to unnecessary complexity and increased gas costs.

```
assert(secretSeed != 0);
assert(s.phase == 1);
assert(randomWords.length == 1);
assert(inputSeed != 0);
assert(sequence < type(uint64).max);
```

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation: Use `assert` statements in Solidity judiciously, reserving them for conditions that indicate critical and unforeseen errors. Avoid overuse to maintain code clarity and prevent unnecessary gas consumption due to these costly checks.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> All of these conditions written with assert statements are verifying responses from the VRF contract. We aim for 100% coverage with Hardhat. We don't believe there's any intentional way to cause these errors. Nevertheless, even though we can't intentionally trigger these errors, they would be extremely critical if they were to occur. That's why we've left them in with assert statements, as a precautionary measure.

F-2024-1503 - Revert String Size Optimization - Info

Description:

Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore` , along with additional overhead to calculate memory offset, etc.

Number Of Instances

There are 22 instances of this issue

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation:

To optimize Gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> Our functions are mostly called through our gRPC. Since we use gRPC for communication, validations are performed off-chain. Consequently, there are very few calls that would revert. We believe that this aspect doesn't need to be applied.

F-2024-1504 - Unneeded initializations of uint256 and bool variable to 0/false - Info

Description: In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing `uint256` variables to `0` and `bool` variables to `false` when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

```
for (uint256 index = 0; index < len; )
for (uint256 i = 0; i < tLen; ) {
for (uint256 i = 0; i < len; ) {
```

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: It is recommended not to initialize integer variables to `0` to and boolean variables to `false` to save some Gas.

Remediation (Revised Commit: 179aa1d): The Nexon team fixed the issue by removing unneeded initializations.

F-2024-1547 - Missing Event Logging - Info

Description:

Events for critical state changes should be emitted for tracking actions off-chain.

It was observed that events are missing in the following function:

- `setKeyHash()`
- `clearSlot()`

Events are crucial for tracking changes on the blockchain, especially for actions that alter significant contract states or permissions. The absence of events in these functions means that external entities, such as user interfaces or off-chain monitoring systems, cannot effectively track these important changes.

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

Consider implementing and emitting events for the necessary functions.

Remediation (Revised Commit: 6576eb9): The Nexon team fixed the issue by defining and emitting the missing events.

F-2024-1548 - Use `calldata` instead of `memory` - Info

Description:

In Solidity, when defining parameters for external functions, the choice between `calldata` and `memory` can significantly impact gas consumption. The `calldata` storage location is a read-only area for function arguments, which exists outside the Ethereum Virtual Machine (EVM) memory and is cheaper to access. Conversely, `memory` denotes a temporary place to store data, and using it for function parameters, especially arrays or structs, requires data to be copied from `calldata` to `memory`, incurring additional gas costs.Vulnerability Description: In Solidity, when defining parameters for external functions, the choice between calldata and memory can significantly impact gas consumption. The calldata storage location is a read-only area for function arguments, which exists outside the Ethereum Virtual Machine (EVM) memory and is cheaper to access. Conversely, memory denotes a temporary place to store data, and using it for function parameters, especially arrays or structs, requires data to be copied from calldata to memory, incurring additional gas costs.

It was observed that `memory` used instead of `calldata` in the following functions:

- `addItem()`
- `addBatchItem()`
- `returnBatchItem()`
- `addItemByChar()`
- `addBatchItemByChar()`
- `addBatchItemByChar()`
- `returnBatchItemByChar()`
- `setSummaryBatch()`
- `reveal()`
- `depositByOwner(string memory reason)`

This inefficiency arises during the `abi.decode()` step for external calls with `memory` arrays, where a loop is needed to copy each element from `calldata` to `memory`. Each iteration of this loop costs a minimum of 60 gas, leading to increased costs proportional to the array's length. Even when arrays are passed to internal functions that may modify them, starting with `calldata` for external functions and converting to `memory` as needed can still result in gas savings, particularly when function modifiers are involved.

The screenshots below demonstrate the gas savings achieved by using `calldata` instead of `memory`.

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Use `calldata` instead of `memory` for the given functions.

Remediation (Revised Commit: 02dc780): The Nexon team fixed the issue by using `calldata` instead of `memory`.

F-2024-1574 - Cache Local Variable Array Length In For Loop - Info

Description: If not cached, the solidity compiler will always read the length of the array during each iteration. If it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

```
for (uint256 i = 0; i < sellerOrders.length; ) {
```

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation: To optimize gas consumption, cache the length of memory arrays before for loop iterations in Solidity. This reduces redundant mload operations, saving 3 gas per iteration after the first and improving overall contract efficiency.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> This causes a 'stack too deep' error.

F-2024-1575 - Revert String Size Optimization - Info

Description:

Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore` , along with additional overhead to calculate memory offset, etc.

Number Of Instances

There are 15 instances of this issue

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation:

To optimize Gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> Our functions are mostly called through our gRPC. Since we use gRPC for communication, validations are performed off-chain. Consequently, there are very few calls that would revert. We believe that this aspect doesn't need to be applied.

F-2024-1576 - Storage Layout Optimization - Info

Description: Storage Layout Optimization in Solidity involves arranging state variables to minimize gas costs. Since storage is expensive, combining variables into as few slots as possible and deleting unneeded variables can significantly reduce the gas needed for contract operations.

```
struct Commission {
    address commissionTo;
    uint256 commissionPercentage;
    uint32 dAppId;
}
```

This struct can be optimized by using like

```
struct Commission {
    uint256 commissionPercentage; //32 bytes
    address commissionTo; //20 bytes
    uint32 dAppId; //4 bytes
}
```

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To optimize storage and reduce gas costs, rearrange the storage variables in a way that makes the most of each 32-byte storage slot.

Remediation (Revised commit: 6ecaf18): The Nexon team fixed the issue by optimizing the `Commission` structure.

F-2024-1577 - Unneeded initializations of uint256 and bool variable to 0/false - Info

Description: In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing `uint256` variables to `0` and `bool` variables to `false` when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

```
for (uint256 i = 0; i < sellerOrders.length; ) {
```

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: It is recommended not to initialize integer variables to `0` to and boolean variables to `false` to save some Gas.

Remediation (Revised commit: 7a3f7e7): The Nexon team fixed the issue by removing unneeded initializations.

F-2024-1578 - Public Functions That Should Be External - Info

Description: Functions that are meant to be exclusively invoked from external sources should be designated as `external` rather than `public`. This is essential to enhance both the gas efficiency and the overall security of the contract.

```
function fillsAmounts(bytes32 orderHash) public view returns (uint256) {
```

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To optimize gas usage and improve code clarity, declare functions that are not called internally within the contract and are intended for external access as `external` rather than `public`. This ensures that these functions are only callable externally, reducing unnecessary gas consumption and potential security risks.

Remediation (Revised commit: 1f54c5c): The Nexon team fixed the issue by changing visibility from `public` to `external`.

F-2024-1580 - Style Guide Violation - Info

Description: The style guide says that, within a contract, the ordering should be 1) Type declarations, 2) State variables, 3) Events, 4) Errors, 5) Modifiers, and 6) Functions, but the contract(s) below do not follow this ordering.

```
9:abstract contract Exchange1155 is Exchange {    // @audit-issue : Order lay
out of this contract is not correct. It should be like this:
    1 -> using SafeERC20
    2 -> _fillsAmounts
    3 -> SELLER_ORDER_TYPEHASH // State variables before events.
    4 -> BUYER_ORDER_TYPEHASH // State variables before events.
    ...
```

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Follow the official style guide.

Remediation (Revised commit: 6b2a773): The Nexon team fixed the issue by following the Solidity guidelines.

F-2024-1609 - Use `calldata` instead of `memory` - Info

Description:

In Solidity, when defining parameters for external functions, the choice between `calldata` and `memory` can significantly impact gas consumption. The `calldata` storage location is a read-only area for function arguments, which exists outside the Ethereum Virtual Machine (EVM) memory and is cheaper to access. Conversely, `memory` denotes a temporary place to store data, and using it for function parameters, especially arrays or structs, requires data to be copied from `calldata` to `memory`, incurring additional gas costs.Vulnerability Description: In Solidity, when defining parameters for external functions, the choice between calldata and memory can significantly impact gas consumption. The calldata storage location is a read-only area for function arguments, which exists outside the Ethereum Virtual Machine (EVM) memory and is cheaper to access. Conversely, memory denotes a temporary place to store data, and using it for function parameters, especially arrays or structs, requires data to be copied from calldata to memory, incurring additional gas costs.

It was observed that `memory` used instead of `calldata` in the following functions:

- `_validateSignature()`
- `_atomicMatch1155()`
- `_tokenExchange()`
- `_cancelOrder1155()`
- `_validateOrder1155()`
- `_hashSellerOrder1155()`
- `_hashBuyerOrder1155()`
- `cancelOrder1155()`
- `atomicMatch721()`
- `atomicMatch1155()`
- `validateSignature()`
- `hashSellerOrder1155()`
- `hashBuyerOrder1155()`

This inefficiency arises during the `abi.decode()` step for external calls with `memory` arrays, where a loop is needed to copy each element from `calldata` to `memory`. Each iteration of this loop costs a minimum of 60 gas, leading to increased costs proportional to the array's length. Even when arrays are passed to internal functions that may modify them, starting with `calldata` for external functions and converting to `memory` as needed can still result in gas savings, particularly when function modifiers are involved.

Assets:

- Marketplace.sol [https://github.com/nexpace-dev/marketplace-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Use `calldata` instead of `memory` for the given functions.

Remediation (Revised Commit: 1ce13e4): The Nexon team fixed the issue by using `calldata` instead of `memory`.

F-2024-1641 - Missing checks for `address(0)` - Info

Description:

In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. The "
Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.
For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

```
_token = IERC20(token_);
_dAppOwner = dAppOwner_;
creatorBeacon = creatorBeacon_;
itemIssuance = itemIssuance_;
neso = neso_;
creatorFactory = creatorFactory_;
blackhole = blackhole_;
_token = token_;
_dAppOwner = newDAppOwner;
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Impact: 1/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: It is strongly recommended to implement checks to prevent the zero address from being set during the initialization of contracts. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Remediation (Revised Commit: e021cf4): The Nexon team fixed the issue by implementing zero address validation for the necessary variables.

F-2024-1642 - Use `Ownable2Step` rather than `Ownable` - Info

Description: Ownable2Step and Ownable2StepUpgradeable prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.

```
contract CommissionForCreator is Ownable {
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation: Consider using `Ownable2Step` or `Ownable2StepUpgradeable` from OpenZeppelin Contracts to enhance the security of your contract ownership management. These contracts prevent the accidental transfer of ownership to an address that cannot handle it, such as due to a typo, by requiring the recipient of owner permissions to actively accept ownership via a contract call. This two-step ownership transfer process adds an additional layer of security to your contract's ownership management.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> All Contracts in **nexpace** transfer ownership to `multisig` contract during the deployment process. Therefore, we believe there is no need to use the `Ownable2Step` contract to prevent incidents of sending ownership to invalid addresses.

F-2024-1643 - Owner Can Renounce Ownership - Info

Description: The smart contract under inspection inherits from the `Ownable` library, which provides basic authorization control functions, simplifying the implementation of user permissions. While the contract allows for the transfer of ownership to a different address or account, it also retains the default `renounceOwnership` function from `Ownable`. Once the owner uses this function to renounce ownership, the contract becomes ownerless. Evidence in the transaction logs shows that, following the activation of the `renounceOwnership` function, any attempts to invoke functions requiring owner permissions fail, with the error message: `"Ownable: caller is not the owner."` This condition makes the contract's adjustable parameters immutable, potentially rendering the contract ineffective for any future administrative modifications that might be needed.

```
contract CommissionForCreator is Ownable {
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation: To mitigate this vulnerability:

- Override the `renounceOwnership` function to revert transactions: By overriding this function to simply revert any transaction, it will become impossible for the contract owner to unintentionally (or intentionally) render the contract ownerless and thus immutable.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> All Contracts in **nexpace** transfer ownership to `multisig` contract during the deployment process. Therefore, we believe there is no need to use the `Ownable2Step` contract to prevent incidents of sending ownership to invalid addresses.

F-2024-1644 - Cache Local Variable Array Length In For Loop - Info

Description: If not cached, the solidity compiler will always read the length of the array during each iteration. If it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

```
for (uint256 i = 0; i < newWeightInfo.length; ) {
for (uint256 i = 0; i < rewards.length; i++) {
for (uint256 i = 0; i < tokens.length; i++) {
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To optimize gas consumption, cache the length of memory arrays before for loop iterations in Solidity. This reduces redundant mload operations, saving 3 gas per iteration after the first and improving overall contract efficiency.

Remediation (Revised commit: 1038bb1): The Nexon team fixed the issue by caching array lengths.

F-2024-1646 - Revert String Size Optimization - Info

Description: Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore` , along with additional overhead to calculate memory offset, etc.

*It was observed that there are **50 instances** of this issue.*

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation: To optimize Gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> Our functions are mostly called through our gRPC. Since we use gRPC for communication, validations are performed off-chain. Consequently, there are very few calls that would revert. We believe that this aspect doesn't need to be applied.

F-2024-1647 - Redundant import - Info

Description: The contracts in `CreatorWallet`

```
import { Initializable } from "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
import { SelfCallUpgradeable } from "@projecta/multisig-contracts/contracts/common/SelfCallUpgradeable.sol";
```

are imported but never used.

This redundancy in import operations has the potential to result in unnecessary gas consumption during deployment and could potentially impact the overall code quality.

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Remove redundant imports, and ensure that the contract is imported only in the required locations, avoiding unnecessary duplications.

Remediation (Revised commit: a931d98): The Nexon team fixed the issue by removing redundant imports.

F-2024-1649 - Unneeded initializations of uint256 and bool variable to 0/false - Info

Description: In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing `uint256` variables to `0` and `bool` variables to `false` when they are not subsequently used in the code can lead to unnecessary gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

```
for (uint256 i = 0; i < newWeightInfo.length; ) {
for (uint256 i = 0; i < item721ContractsLength; ) {
for (uint256 i = 0; i < rewards.length; i++) {
for (uint256 i = 0; i < tokens.length; i++) {
for (uint256 i = 0; i < toLength; i++) {
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: It is recommended not to initialize integer variables to `0` to and boolean variables to `false` to save some Gas.

Remediation (Revised commit: 05105d6): The Nexon team fixed the issue by removing unneeded initializations.

F-2024-1682 - Conditions Can Be Optimized - Info

Description: In Solidity development, the way conditional statements are written can significantly impact the contract's gas efficiency, readability, and overall maintainability. Complex or poorly structured conditions can lead to increased execution costs and potential security vulnerabilities. There is often an opportunity to simplify and streamline these statements. By optimizing conditional logic, developers can create smarter contracts that are not only more cost-effective in terms of gas usage but also more secure and easier to understand and maintain. This optimization plays a crucial role in enhancing the performance and reliability of smart contracts on the Ethereum blockchain.

```
_vrfRequesters[requesterAddress].isVRFRequester == false,
_vrfRequesters[requesterAddress].isVRFRequester == true,
```

Assets:

- VRFManager.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/hacken-final]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Optimize conditional statements in Solidity for better gas efficiency, readability, and maintainability. Simplify logic where possible and consider the cost implications of each condition to create more effective and secure smart contracts.

Remediation (Revised Commit: 4944bab): The Nexon team fixed the issue by optimizing the conditional statements.

F-2024-1684 - Unused Error Definition - Info

Description: The error is declared, but never used.

```
Path: ./contracts/VRF/chainlink/VRFConsumerBaseV2Plus.sol

error ZeroAddress();
```

Deploying contracts on the Ethereum network incurs gas costs proportional to the contract's bytecode size. Redundant code, including unused error declarations, increases the contract's size unnecessarily, leading to higher deployment cost.

This unused error results in redundant logic within the code.

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Unused `error` definitions should be removed from the contract, and if needed, consolidated into a separate file to avoid duplication.

Remediation (Revised Commit: a8225e1): The Nexon team fixed the issue by removing the unused error.

F-2024-1685 - Increments can be `unchecked` in for-loops - Info

Description: Newer versions of the Solidity compiler will check for integer overflows and underflows automatically. This provides safety but increases gas costs.
When an unsigned integer is guaranteed to never overflow, the unchecked feature of Solidity can be used to save gas costs.A common case for this is for-loops using a strictly-less-than comparision in their conditional statement.

```
for (uint256 i = 0; i < rewards.length; i++) {
for (uint256 i = 0; i < tokens.length; i++) {
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Use unchecked math to block overflow / underflow check to save Gas where overflow / underflow findings are unrealistic.

Remediation (Revised Commit: 041ea50): The Nexon team fixed the issue by implementing incrementation for the variable `i` within an unchecked block.

F-2024-1705 - Missing Functionality - Info

Description: That 2 function does not have any logic implemented in them and they are just emitting. If emitting those values are important for offchain, then those emits should be done in function where adding operation is really happening.

```
function addItem721(uint64 sequence, uint24 universe, bytes calldata itemHash
) external whenExecutable {
    emit Item721Added(sequence, universe, itemHash);
}

function addItem1155(uint64 sequence, uint24 universe, bytes calldata itemHas
h) external whenExecutable {
    emit Item1155Added(sequence, universe, itemHash);
}
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Mitigated

Classification

Severity: Info

Recommendations

Remediation: Either implement a functionality to those functions or remove them from the contract.

Remediation (Mitigated): The Nexon team mitigated the issue with the following statement;

> These events are for verification on offchain logic. so it is correct that no functionality.

F-2024-1706 - Add `unchecked {}` For Subtractions - Info

Description: `unchecked {}` keyword can be added for subtractions where the operands cannot underflow because of a previous `require()` or `if`-statement. Example scenario:
`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a }`

```
Path: ./contracts/ItemIssuance/ItemIssuance.sol
require(amount <= _universes[universe - 1].item721Amount, "ItemIssuance/invalidAmount: too large amount");
    require(0 < amount, "ItemIssuance/invalidAmount: requested amount must be bigger than 0");
...
...
_universes[universe - 1].item721Amount -= amount;
```

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: In scenarios where subtraction cannot result in underflow due to prior `require()` or `if`-statements, wrap these operations in an `unchecked` block to save gas. This optimization should only be applied when the safety of the operation is assured. Carefully analyze each case to confirm that underflow is impossible before implementing `unchecked` blocks, as incorrect usage can lead to vulnerabilities in the contract.

Remediation (Revised commit: 7c9fb41): The Nexon team fixed the issue by moving the subtractions to the inside of unchecked block.

F-2024-1738 - Missing Event Logging - Info

Description:

Events for critical state changes should be emitted for tracking actions off-chain.

It was observed that events are missing in the following function:

- `update()`

Events are crucial for tracking changes on the blockchain, especially for actions that alter significant contract states or permissions. The absence of events in these functions means that external entities, such as user interfaces or off-chain monitoring systems, cannot effectively track these important changes.

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

Consider implementing and emitting event for the necessary function.

Remediation (Revised Commit: 098604e): The Nexon team fixed the issue by defining and emitting an event.

F-2024-1739 - Unsafe Usage of transferFrom() - Info

Description:

The smart contracts under review employ the `ERC721.transferFrom()` method for transferring Non-Fungible Tokens (NFTs).

```
function transferERC721(address token, address to, uint256 tokenId) external
isSelfCall {
    IERC721(token).transferFrom(address(this), to, tokenId);
}
```

While this method successfully transfers NFTs between addresses, it omits critical safety checks that are inherent to the `safeTransferFrom()` method. Specifically, `transferFrom()` does not verify whether the recipient address is capable of receiving NFTs, which is a crucial step to ensure the safe handling of NFTs, especially when interacting with smart contracts. The absence of such safety mechanisms in the current implementation can lead to scenarios where NFTs are transferred to contracts that are not designed to handle them. This oversight can result in NFTs being permanently locked within contracts, rendering them inaccessible and effectively lost.

Assets:

- Commission.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

- **Use safeTransferFrom() for Transfers:** Replace **transferFrom()** with **safeTransferFrom()** for all NFT transfers. This ensures that the receiving contract is capable of handling ERC721 tokens, preventing accidental loss.
- **Implement Checks-Effects-Interactions Pattern:** When integrating the recommended changes, ensure that the contract's functions adhere to the checks-effects-interactions pattern to prevent potential reentrancy attacks. This involves performing all necessary conditions and state changes before interacting with external contracts.

Remediation (Revised Commit: 098604e): The Nexon team fixed the issue by implementing `safeTransferFrom` instead of `transferFrom`.

F-2024-1803 - The `constructor` function lacks parameter validation - Info

Description: Constructors and initialization functions play a critical role in contracts by setting important initial states when the contract is first deployed before the system starts. The parameters passed to the constructor and initialization functions directly affect the behavior of the contract / protocol. If incorrect parameters are provided, the system may fail to run, behave abnormally, be unstable, or lack security. Therefore, it's crucial to carefully check each parameter in the constructor and initialization functions. If an exception is found, the transaction should be rolled back.

```
constructor(
    address trustedForwarder,
    IVRFManager vrfManager_,
    bytes32 keyHash_,
    uint64 maxDepth_
) ERC2771Context(trustedForwarder) VRFRequester(vrfManager_) {
    _keyHash = keyHash_;
    _state.maxDepth = maxDepth_;
}
```

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Incorporate comprehensive parameter validation in your contract's constructor and initialization functions. This should include checks for value ranges, address validity, and any other condition that defines a parameter as valid. Use `require` statements for validation, providing clear error messages for each condition. If any validation fails, the `require` statement will revert the transaction, preventing the contract from being deployed or initialized with invalid parameters.

Remediation (Revised Commit: 107bc88): The Nexon team fixed the issue by validating the constructor parameter and the function to

change the `VRFManager` address.

F-2024-1804 - `Private` functions only called once can be inlined to save gas - Info

Description: If a private function is only used once, there is no need to modularize it, unless the function calling it would otherwise be too long and complex. Not inlining costs 20 to 40 gas because of two extra JUMP instructions and additional stack operations needed for function calls.

```
function next() external whenExecutable {
   _next();
}
..
function _next() private {
```

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Inline `private` functions in Solidity that are called only once to save gas. This avoids the additional gas cost of 20 to 40 units associated with extra JUMP instructions and stack operations required for separate function calls, unless the calling function becomes too complex.

Remediation (Revised Commit: be01c84): The Nexon team fixed the issue by implementing the `_next()` function logic instead of calling it.

F-2024-1806 - Style guide: Function ordering does not follow the Solidity style guide - Info

Description: According to the Solidity style guide, functions should be laid out in the following order : `constructor()`, `receive()`, `fallback()`, `external`, `public`, `internal`, `private`, but the cases below do not follow this pattern.

```
164:    function _next() private
..
183:    function _msgSender() internal
```

Assets:

- Collection.sol [https://github.com/nexpace-dev/msn-contract]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Follow the official Solidity guidelines in order to correct this observation.

Remediation (Revised commit: 176fb1b): The Nexon team fixed the issue by following the Solidity guidelines.

F-2024-4573 - Cache Variable Array Length In For Loop - Info

Description: Failing to cache the array length when iterating through arrays in Solidity can have significant performance and Gas cost implications. In Solidity, array lengths can change during execution due to external calls or storage modifications. When the array length is not cached before entering a loop, it is recomputed with each iteration, leading to unnecessary Gas consumption.

```
for (uint256 i; i < buyerOrders.length; ) {
```

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To enhance performance and reduce Gas costs, cache the array length before entering a for loop in Solidity. This approach prevents repeated computation of the array length and mitigates the risk of reentrancy attacks due to array length changes during loop execution.

An example implementation for variable caching:

```
uint256 buyerOrdersLength = buyerOrders.length;
for (uint256 i; i < buyerOrdersLength; ) {
```

Remediation The Nexon team fixed the issue in the commit: `8226995` with caching the array length.

F-2024-4575 - Public Functions That Should Be External - Info

Description: Functions that are meant to be exclusively invoked from external sources should be designated as `external` rather than `public`.

```
function fillsAmounts(bytes32 orderHash) public view returns (uint256) {
```

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To optimize Gas usage and improve code clarity, declare functions that are not called internally within the contract and are intended for external access as `external` rather than `public`. This ensures that these functions are only callable externally, reducing unnecessary gas consumption and potential security risks.

Remediation The Nexon team fixed the issue in the commit: `cd6fafc` by changing the function visibility.

F-2024-5426 - Missing Checks For Zero Address - Info

Description:

In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. The "
Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.
For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

```
_checkPoint = checkPoint_;
_initOwner = initOwner_;
_erc20BridgeTokenImpl = erc20BridgeTokenImpl_;
_erc721BridgeTokenImpl = erc721BridgeTokenImpl_;
_erc1155BridgeTokenImpl = erc1155BridgeTokenImpl_;
...
_blackhole = blackhole_;
_NXPCAmountManager = INXPCAmountManager(NXPCAmountManager_);
...
_equip = IEquip(equip_);
_NXPCAmountManager = INXPCAmountManager(NXPCAmountManager_);
_vault = vault_;
```

Assets:

- DestinationBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]
- SourceBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]
- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts]
- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Fixed

Classification

Impact: 1/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: It is strongly recommended to implement checks to prevent the zero address from being set during the initialization of contracts. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Remediation The Nexon team fixed the issue in the commit: `c7d0932` with adding zero address controls.

F-2024-5427 - Cache Variable Array Length In For Loop - Info

Description: Failing to cache the array length when iterating through arrays in Solidity can have significant performance and Gas cost implications. In Solidity, array lengths can change during execution due to external calls or storage modifications. When the array length is not cached before entering a loop, it is recomputed with each iteration, leading to unnecessary Gas consumption.

```
for (uint256 i; i < tokenIds.length; ) {
for (uint256 i = 0; i < slotLength.length; ) {
```

Assets:

- DestinationBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]
- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To enhance performance and reduce Gas costs, cache the array length before entering a for loop in Solidity. This approach prevents repeated computation of the array length and mitigates the risk of reentrancy attacks due to array length changes during loop execution.

Example implementation of caching variables:

```
uint256 tokenIdsLength = tokenIds.length;
for (uint256 i; i < tokenIdsLength ; ) {
...
uint256 slotLengthLength = slotLength.length;
for (uint256 i = 0; i < slotLengthLength; ) {
```

Remediation The Nexon team fixed the issue in the commit: `3342be1` with caching the array length.

F-2024-5428 - Conditions Can Be Optimized - Info

Description:

In Solidity development, the way conditional statements are written can significantly impact the contract's gas efficiency, readability, and overall maintainability. This optimization plays a crucial role in enhancing the performance and reliability of smart contracts on the Ethereum blockchain.

```
require(getBlocklist(from) == false, "BridgeToken/blocklist: msg sender is a blocklist")
...
require(_isBlocklist[_msgSender()] == false, "DestinationBridge/blocklist: msg sender is a blocklist");
if ((_isBlocklist[recipient] == true) || (_isBlocklist[sender] == true)) {
if ((_isBlocklist[recipient] == true) || (_isBlocklist[sender] == true)) {
if ((_isBlocklist[recipient] == true) || (_isBlocklist[sender] == true)) {
...
require(_isBlocklist[_msgSender()] == false, "SourceBridge/blocklist: msg sender is a blocklist");
if ((_isBlocklist[recipient] == true) || (_isBlocklist[sender] == true))
if ((_isBlocklist[recipient] == true) || (_isBlocklist[sender] == true)) {
require(isBridgedNft == true, "SourceBridge/wrongAmount: insufficient balance");
if ((_isBlocklist[recipient] == true) || (_isBlocklist[sender] == true)) {
```

Assets:

- ERC20BridgeToken.sol [https://github.com/nexpace-dev/nexpace-contracts]
- ERC721BridgeToken.sol [https://github.com/nexpace-dev/nexpace-contracts]
- ERC1155BridgeToken.sol [https://github.com/nexpace-dev/nexpace-contracts]
- DestinationBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]
- SourceBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Optimize conditional statements in Solidity to enhance gas efficiency, readability, and maintainability. Simplify the logic wherever possible.

For instance, consider changing the following code:

```
if ((_isBlocklist[recipient] == true) || (_isBlocklist[sender] == true)) {
```

to:

```
if ((_isBlocklist[recipient]) || (_isBlocklist[sender])) {
```

This adjustment eliminates unnecessary comparisons, leading to more efficient code.

Remediation The Nexon team fixed the issue in the commit: `e94c07d` with removing unnecessary variable controls.

F-2024-5429 - Revert String Size Optimization - Info

Description:

Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore` , along with additional overhead to calculate memory offset.

Example:

```
require(msg.sender == _bridgeContract, "BridgeToken/unauthorized: unauthorize
d");

require(getBlocklist(from) == false, "BridgeToken/blocklist: msg sender is a
blocklist");
...
```

Number of instances: 110

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation:

To optimize Gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

Remediation (Accepted): The Nexon team accepted the issue.

F-2024-5431 - State Variables Only Set in The Constructor Should Be Declared as Immutable - Info

Description:

Compared to regular state variables, the Gas costs of constant and immutable variables are much lower. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed.

The following variables can be declared `immutable`:

```
address private _initOwner;
IERC20BridgeToken private _erc20BridgeTokenImpl;
IERC721BridgeToken private _erc721BridgeTokenImpl;
IERC1155BridgeToken private _erc1155BridgeTokenImpl;
```

Assets:

- DestinationBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

Consider declaring the aforementioned state variables as immutable if there is not intention to change them in the future.

Remediation The Nexon team fixed the issue in the commit: `f667175` with declaring those variables as immutable.

F-2024-5434 - Storage Layout Optimization - Info

Description:

Storage Layout Optimization in Solidity involves arranging state variables to minimize Gas costs. Since storage is expensive, combining variables into as few slots as possible and deleting unneeded variables can significantly reduce the Gas needed for contract operations.

```
contract NXPCDistributor is ERC2771Context, NextOwnablePausable {
    /// @notice Equip contract address
    IEquip private immutable _equip;

    /* solhint-disable var-name-mixedcase */
    /// @notice NXPCAmountManager contract address
    INXPCAmountManager private immutable _NXPCAmountManager;
    /* solhint-enable var-name-mixedcase */

    address private _vault;

    /// @notice Current round number
    uint256 private _currentRound;

    /// @notice Current status
    bool private _isStarted;

    /// @notice Basket merkle root: keccak256(keccak256(round || itemId || sl
otLength))
    mapping(uint256 => bytes32) private _basketMerkleRoot;
    ...
    ...
```

The layout can be optimized by moving `_isStarted` next to `_vault`:

```
    // Rest of the definitions
    address private _vault;

    /// @notice Current status
    bool private _isStarted;

    // Rest of the definitions
```

Assets:

- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To optimize storage and reduce Gas costs, rearrange the storage variables in a way that makes the most of each 32-byte storage slot.

Remediation The Nexon team fixed the issue in the commit: `fb24278` by moving `_isStarted` next to `_vault`.

F-2024-5436 - Revert String Size Optimization - Info

Description: Shortening the revert strings to fit within 32 bytes will decrease deployment cost and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore` , along with additional overhead to calculate memory offset.

```
require(deployed != address(0), "MinProxy/zeroAddress: create failed");
```

Assets:

- MinProxy.sol [https://github.com/nexpace-dev/min-proxy]

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: To optimize Gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

Remediation (Accepted): The Nexon team accepted the issue.

F-2024-5436 - Revert String Size Optimization - Info

Description: Shortening the revert strings to fit within 32 bytes will decrease deployment cost and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore` , along with additional overhead to calculate memory offset.

```
require(deployed != address(0), "MinProxy/zeroAddress: create failed");
```

Assets:

- MinProxy.sol [https://github.com/nexpace-dev/min-proxy]

Status: Accepted

Classification

Impact: 1/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: To optimize Gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of Gas used during deployment and runtime when the revert condition is met.

Remediation (Accepted): The Nexon team accepted the issue.

F-2024-5443 - Potential Subtraction Operations Optimisation - Info

Description: Use the `unchecked` keyword for subtractions where operands cannot underflow due to prior validation with a `require()` statement or an `if` condition. Applying `unchecked` in such scenarios can enhance Gas efficiency by bypassing Solidity's default overflow and underflow checks.

```
bridgedBalances[sourceBlockchainID][sourceBridgeAddress][sourceTokenAddress]
= currentBalance - amount;
...
bridgedFts[sourceBlockchainID][sourceBridgeAddress][sourceTokenAddress][token
Ids[i]] = currentBalance - amounts[i];
```

Assets:

- SourceBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: In scenarios where subtraction cannot result in underflow due to prior `require()` or `if`-statements, wrap these operations in an `unchecked` block to save gas. This optimization should only be applied when the safety of the operation is assured. Carefully analyze each case to confirm that underflow is impossible before implementing `unchecked` blocks, as incorrect usage can lead to vulnerabilities in the contract.

Remediation The Nexon team fixed the issue in the commit: `b2984b4` with adding unchecked keyword.

F-2024-5444 - Constants Declared as Public - Info

Description:

In Solidity, constants represent immutable values that cannot be changed after they are set at compile-time. By default, constants have internal visibility, meaning these can be accessed within the contract they are declared in and in derived contracts. If a constant is explicitly declared as `public`, Solidity automatically generates a getter function for it. While this might seem harmless, it actually incurs a Gas overhead, especially when the contract is deployed, as the EVM needs to generate bytecode for that getter. Conversely, declaring constants as `private` ensures that no additional getter is generated, optimizing Gas usage.

```
uint256 public constant UNLOCK_BRIDGE_TOKENS_REQUIRED_GAS = 200_000;
address public constant WARP_PRECOMPILE_ADDRESS = 0x0200000000000000000000000000000000000005;
...
address public constant NATIVE_TOKEN = 0x0200000000000000000000000000000000000001;
address public constant WARP_PRECOMPILE_ADDRESS = 0x0200000000000000000000000000000000000005;
uint256 public constant CREATE_BRIDGE_TOKENS_REQUIRED_GAS = 3_000_000;
uint256 public constant MINT_BRIDGE_TOKENS_REQUIRED_GAS = 200_000;
```

Assets:

- DestinationBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]
- SourceBridge.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Accepted

Classification

Severity: Info

Recommendations

Remediation: To optimize Gas usage in your Solidity contracts, declare constants with `private` visibility rather than `public` when possible. Using `private` prevents the automatic generation of a getter function, reducing Gas overhead, especially during contract deployment.

Remediation The Nexon team accepted the issue.

F-2024-5446 - Public Functions That Should Be External - Info

Description:

Functions that are meant to be exclusively invoked from external sources should be designated as `external` rather than `public`. This is essential to enhance both the Gas efficiency and the overall security of the contract.

The following functions can be defined as `external`:

- `issuanceCycleInfo`
- `addingItemData`

Assets:

- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation:

To optimize Gas usage and improve code clarity, declare functions that are not called internally within the contract and are intended for external access as `external` rather than `public`. This ensures these functions are only callable externally, reducing unnecessary Gas consumption and potential security risks.

Remediation The Nexon team fixed the issue in the commit: `d4934cc` with changing the function visibilities.

F-2024-5447 - Style Guide Violation - Info

Description:

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering. These guidelines help to maintain a consistent structure across different contracts, libraries, or interfaces, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each contract, library, or interface is as follows:

1. Type declarations
2. State variables
3. Events
4. Errors
5. Modifiers
6. Functions

Functions should be ordered and grouped by their visibility as follows:

1. Constructor
2. Receive function (if exists)
3. Fallback function (if exists)
4. External functions
5. Public functions
6. Internal functions
7. Private functions

Within each grouping, `view` and `pure` functions should be placed at the end.

Furthermore, following the Solidity naming convention and adding NatSpec annotations for all functions are strongly recommended. These measures aid in the comprehension of code and enhance overall code quality.

```
    function _msgSender() internal view virtual override(Context, ERC2771Cont
ext) returns (address sender) {
        return ERC2771Context._msgSender();
    }

    function _msgData() internal view virtual override(Context, ERC2771Contex
t) returns (bytes calldata) {
        return ERC2771Context._msgData();
    }
```

These two internal functions should be defined before the private functions.

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: Consistent adherence to the official Solidity style guide is recommended. This enhances readability and maintainability of the code, facilitating seamless interaction with the contracts. Providing comprehensive NatSpec annotations for functions and following Solidity's naming conventions further enrich the quality of the code.

Remediation The Nexon team fixed the issue in the commit: `c825302` by following the guide.

F-2025-9634 - validAddress Modifier Prevents Using address(0x0) as Blackhole Address - Info

Description:

The `validAddress()` modifier was designed to prevent the setting of zero addresses for crucial protocol addresses such as `NXPCAccountManager` and `trustedForwarder`, among others.

However, some contracts also initialize another important address for burning NFTs or other assets. For this specific purpose, the `Blackhole` variable is used. Typically, the zero address is designated as a blackhole address to facilitate the burning of assets.

This modifier prevents the setting of the zero address as a blackhole address across the protocol.

ItemIssuance.sol:

```
modifier validAddress(address addr) {
    require(addr != address(0), "ItemIssuance/invalidAddress: couldn't be zer
o address");
    _;
}
```

ItemIssuance.sol:

```
constructor(
    address trustedForwarder,
    address _blackhole,
    address _NXPCAmountManager
)
    ERC2771Context(trustedForwarder)
    validAddress(trustedForwarder)
    validAddress(_blackhole) // @audit-issue - blackhole address can be 0x000
0000000000000000000000000000000000000
    validAddress(_NXPCAmountManager)
{
```

Assets:

- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Accepted

Classification

Impact:	1/5
Likelihood:	5/5
Exploitability:	Dependent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation: Consider eliminating the `validAddress()` modifier for `_blackhole` constructor variable.

Resolution: This finding was **acknowledged** by the client. In addition, the following statement was provided:

> Since we actually need to reuse NXPC rather than burn it, we plan to set the blackhole address to a multisig address. Therefore, the blackhole address should not be `0x000...000`.

F-2025-9636 - Missing Sanity Check May Cause Unexpected Reverts with Panic Error - Info

Description:

The code below lacks a critical sanity check before subtracting `request.itemAmount` from `universes[universe - 1].poolAmount`. If `request.itemAmount` exceeds the available `poolAmount`, this line will cause an underflow and revert with a generic panic error (`Panic(0x11)`), which makes it difficult for developers and users to diagnose the actual cause of failure.

ItemIssuance.sol:

```
if (isConfirmed) {
    request.status = Status.CONFIRMED;
    target = blackhole;

    INXPCAmountManager(NXPCAmountManager).addBurnedAmount(request.nxpcAmount)
;
    universes[universe - 1].poolAmount -= request.itemAmount; // @audit-issue
- due to missing error handling (universes[universe - 1].poolAmount >= reques
t.itemAmount), it will revert with panic.

    emit RequestConfirmed(universe, requestId);
}
```

Without a clear `require` statement or custom error, the transaction failure lacks clarity and could complicate debugging and monitoring efforts, especially in production environments where logs and error messages are essential for rapid incident response.

Assets:

- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Mitigated

Classification

Impact: 1/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Introduce an explicit `require` statement to ensure `poolAmount` is sufficient before subtracting:

```
require(universes[universe - 1].poolAmount >= request.itemAmount, "ItemPool/insufficientPoolAmount");
```

Resolution: This finding is **mitigated** by the client with the following statement:

> The withdrawal of quantities from the Pool is entirely managed off-chain.
> This means that items are issued in an off-chain state, not on-chain.
> If the requested quantity does not match the off-chain records, the call will not proceed with a `confirm` state—it will be called with `false`.

F-2025-9637 - Incomplete Function Logic Leads to Discrepancy Between Implementation and Developer Intention - Info

Description:

The `addItem721BaseAmount()` function is declared with a developer note suggesting that it **increases the base amount** of an ERC721 item. However, the implementation merely emits an event without modifying any state or performing the intended logic. This creates a **critical mismatch** between the documentation and the behavior of the contract.

Such incomplete logic can lead to significant confusion for developers, integrators, and auditors who rely on NatSpec comments and emitted events to understand the function's effects. In addition, emitting events without corresponding state changes can mislead off-chain systems or observers into believing that an action has occurred when it has not.

ItemIssuance.sol:

```
/// @notice Increases the base amount of an item within the specified univers
e for ERC721 tokens
/// @param itemId The ID of the item to increase the base amount for
/// @param newLimitSupply The amount to add to the base amount of the item
function addItem721BaseAmount(
    uint256 itemId,
    uint256 newLimitSupply
) external whenUniverseExists(universeOfItem721[_msgSender()]) {
    emit ItemBaseAmountAdded(universeOfItem721[_msgSender()], _msgSender(), i
temId, newLimitSupply);
}
```

Assets:

- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Mitigated

Classification

Impact: 1/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Either update the documentation to reflect the function’s limited purpose or implement the full behavior as documented.

Resolution: The finding is **mitigated** with the following statement:

> the base amount is in offchain. the server catch this event and increase base amount in the database.

F-2025-9640 - Deposited Equipment Items Are Not Moved to Item Pool After Round Ends, Contrary to Documentation - Info

Description:

According to the project's official documentation, once a round ends, **all deposited equipment items are expected to be moved to the Item Pool**, where they can later be re-issued by the `ItemIssuance` contract.

However, based on the current implementation, this expected behavior is not enforced in the code. The `_deposit()` function only transfers NFTs from the user to a `vault()` address, and no subsequent action is taken to transfer them into the Item Pool, either at the end of a round or upon meeting any specific condition.

This discrepancy between documentation and implementation can lead to misaligned assumptions by integrators, users, or auditors relying on the documented behavior.

NXPCDistributor.sol:

```
function _deposit(uint256 round, uint256 tokenId, address user, uint256 slotL
ength) private {
    uint64 itemId = equip.tokenItemId(tokenId);

    require(!isFullSlot(currentRound(), itemId), "NXPCDistributor/invalidSlot
: current item's slot is full");
    unchecked {
        _currentSlotLength[round][itemId]++;
    }
    _currentDepositor[round][tokenId] = user;

    if (currentSlotLength(round, itemId) == slotLength) {
        _isFullSlot[round][itemId] = true;
        _addCurrentBasketLength(round);
    }

    //slither-disable-next-line arbitrary-send-erc20
    equip.transferFrom(user, vault(), tokenId);

    emit Deposit(round, tokenId, itemId, user);
}
```

Assets:

- NXPCDistributor.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

- ItemIssuance.sol [https://github.com/nexpace-dev/nexpace-contracts/tree/feature/25-hacken-audit]

Status: Mitigated

Classification

Impact: 1/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: If the current implementation is intentional, **update the documentation** to reflect the correct item flow to avoid misleading integrators and users.

Resolution: This finding is **mitigated** by the client with the following statement:

> The item pool is an abstract concept and exists both on-chain and off-chain.
> The on-chain item pool refers to the items stored in the Vault.

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. Definitions

Severities

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.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project’s performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

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

Scope Details	
Repository	https://github.com/NEXPACE-Limited/min-proxy ; https://github.com/NEXPACE-Limited/util-contracts ; https://github.com/NEXPACE-Limited/nexpace-contracts ; https://github.com/NEXPACE-Limited/marketplace-contract ; https://github.com/NEXPACE-Limited/maple-contract ; https://github.com/NEXPACE-Limited/msn-contract ; https://github.com/NEXPACE-Limited/multisig-contract
Commit	1af3f49 ; 0e51eaa ; b65662b ; 7033807 ; 5f83de2 ; 08809ac ; 91526df
Whitepaper	https://audits.hacken.io/methodologies/smart-contract-audit-methodology/

Asset	Type
AdminController.sol [https://github.com/NEXPACE-Limited/maple-contract]	Smart Contract
Collection.sol [https://github.com/NEXPACE-Limited/msn-contract]	Smart Contract
Commission.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
CommissionForCreator.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
contracts/access/NextOwnable.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract
contracts/access/NextOwnablePausable.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract
contracts/approve/ApproveControlled.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract
contracts/approve/ApproveController.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract
contracts/approve/token/ERC1155ApproveControlled.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract
contracts/approve/token/ERC20ApproveControlled.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract
contracts/approve/token/ERC721ApproveControlled.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract
contracts/exec/Exec.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract
contracts/exec/interfaces/IExec.sol [https://github.com/NEXPACE-Limited/util-contracts]	Smart Contract

Asset	Type
CreatorFactory.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
CreatorTokenControllerUpgradeable.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
CreatorWallet.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
CreatorWalletLogicUpgradeable.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
DAppRewardAllocationWallet.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
DestinationBridge.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
ERC1155BridgeToken.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
ERC20BridgeToken.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
ERC721BridgeToken.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
Exchange.sol [https://github.com/NEXPACE-Limited/marketplace-contract]	Smart Contract
Exchange1155.sol [https://github.com/NEXPACE-Limited/marketplace-contract]	Smart Contract
Exchange721.sol [https://github.com/NEXPACE-Limited/marketplace-contract]	Smart Contract
ExecutorManager.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
ExecutorManagerUpgradeable.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
ItemIssuance.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
MaplestoryCharacter.sol [https://github.com/NEXPACE-Limited/maple-contract]	Smart Contract
MaplestoryCharacterInventoryImpl.sol [https://github.com/NEXPACE-Limited/maple-contract]	Smart Contract
MaplestoryConsume.sol [https://github.com/NEXPACE-Limited/maple-contract]	Smart Contract
MaplestoryEquip.sol [https://github.com/NEXPACE-Limited/maple-contract]	Smart Contract
Marketplace.sol [https://github.com/NEXPACE-Limited/marketplace-contract]	Smart Contract
MinProxy.sol [https://github.com/NEXPACE-Limited/min-proxy]	Smart Contract

Asset	Type
MSNEquipSummary.sol [https://github.com/NEXPACE-Limited/msn-contract]	Smart Contract
MSUContentsMSUContentsCommission.sol [https://github.com/NEXPACE-Limited/msn-contract]	Smart Contract
MulticallWallet.sol [https://github.com/NEXPACE-Limited/maple-contract]	Smart Contract
Multisig.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
MultisigUpgradeable.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
NextForwarder.sol [https://github.com/NEXPACE-Limited/maple-contract]	Smart Contract
NextMeso.sol [https://github.com/NEXPACE-Limited/maple-contract	Smart Contract
NXPCAmountManager [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
NXPCClaim.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
NXPCDistributor.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
OrderBook.sol [https://github.com/NEXPACE-Limited/marketplace-contract]	Smart Contract
OrderMatch.sol [https://github.com/NEXPACE-Limited/marketplace-contract]	Smart Contract
OwnerManager.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
OwnerManagerUpgradeable.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
RandomSeedGenerator.sol [https://github.com/NEXPACE-Limited/msn-contract]	Smart Contract
SelfCall.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
SelfCallUpgradeable.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
SourceBridge.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
Teller.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
TokenHolder.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract
TokenHolderUpgradeable.sol [https://github.com/NEXPACE-Limited/multisig-contract]	Smart Contract

Asset	Type
VRFManager.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract
VRFRequester.sol [https://github.com/NEXPACE-Limited/nexpace-contracts]	Smart Contract