Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2026.08.04, the SlowMist security team received the CardBox team's security audit application for CardBox, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.

Level	Description
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit
		Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

CardPool is the NFT card pool and pack purchase settlement contract. RepurchaserVault is the repurchaser contract wallet.

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Restricted repurchase seizes NFT without payment; canRepurchase view misleads	Design Logic Audit	Medium	Acknowledged
N2	Repurchase deadline cannot be re-armed during structural lock-out	Design Logic Audit	Low	Acknowledged
N3	Unprivileged holder can clear token compliance restriction via repurchase()	Authority Control Vulnerability Audit	Medium	Acknowledged
N4	Privileged roles lack secondary controls, enabling single-key fund extraction	Authority Control Vulnerability Audit	Medium	Acknowledged
N5	Effects written after ERC721 callback on five paths (CEI violation)	Unsafe External Call Audit	Information	Fixed
N6	Initialization and deprecated storage hygiene issues	Others	Information	Acknowledged
N7	Repurchase payout has no solvency guarantee	Design Logic Audit	Suggestion	Acknowledged

NO	Title	Category	Level	Status
N8	EIP-712 signature authorization incomplete	Authority Control Vulnerability Audit	Suggestion	Acknowledged
N9	cardBoxTemplateIdBy Token not reset after repurchase settlement	Design Logic Audit	Suggestion	Fixed
N10	Non-upgradeable ReentrancyGuard inherited by proxied contract	Others	Suggestion	Acknowledged
N11	DOMAIN_SEPARATOR cached at initialize, stale after chain-id change	Replay Vulnerability	Suggestion	Fixed

4 Code Overview

4.1 Contracts Description

Initial audit version: cardbox-contracts-src-0c8103411.zip:

268b45c29eca7181b9da4359a47df043d3bca45b6a77eccad40e6f3710814ccf

Final audit version: SHA256 (cardbox-contracts-src-20260823.zip) =

98ce113876222d1a38e8884a7a2f56c580a0200e413155fbf99712354363bb37

File Scope:

```
./src
├── CardPool.sol
└── RepurchaserVault.sol
```

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

CardPool			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
depositCard	External	Can Modify State	onlyRole
batchDepositCards	External	Can Modify State	onlyRole
transferFromPool	External	Can Modify State	onlyRole nonReentrant
batchTransferFromPool	External	Can Modify State	onlyRole nonReentrant
purchasePack	External	Can Modify State	nonReentrant onlyRole
purchasePackTrusted	External	Can Modify State	nonReentrant onlyRole
repurchase	External	Can Modify State	nonReentrant
batchRepurchase	External	Can Modify State	nonReentrant
repurchaseCardBox	External	Can Modify State	nonReentrant
batchRepurchaseCardBox	External	Can Modify State	nonReentrant
getRepurchaseInfo	External	-	-
canRepurchase	External	-	-

CardPool			
getRepurchaseStatus	External	-	-
getNonce	External	-	-
getDomainSeparator	External	-	-
setVault	External	Can Modify State	onlyRole
setPlatformRepurchaseVault	External	Can Modify State	onlyRole
setCardManager	External	Can Modify State	onlyRole
markTrustedRequestsConsumed	External	Can Modify State	onlyRole
setDeadlineInterval	External	Can Modify State	onlyRole
setTreasury	External	Can Modify State	onlyRole
onERC721Received	Public	-	-
_depositCard	Internal	Can Modify State	-
_transferFromPool	Internal	Can Modify State	-
_updateRepurchaseInfo	Internal	Can Modify State	-
_purchasePack	Internal	Can Modify State	-
_consumeTrustedRequest	Internal	Can Modify State	-
_settleTrustedPurchase	Internal	Can Modify State	-
_executeTrustedPurchase	Internal	Can Modify State	-
_verifyPurchaseSignature	Internal	Can Modify State	-
_computeStructHash	Internal	-	-
_validateNFTTransfers	Internal	-	-

CardPool			
_transferNFTs	Internal	Can Modify State	-
_validateCardBoxPrizes	Internal	-	-
_mintCardBoxPrizes	Internal	Can Modify State	-
_emitPurchaseCompleted	Internal	Can Modify State	-
_emitTrustedPurchaseCompleted	Internal	Can Modify State	-
_processPayment	Internal	Can Modify State	-
_processPaymentTrusted	Internal	Can Modify State	-
_processPaymentInternal	Internal	Can Modify State	-
_validatePaymentAmounts	Internal	-	-
_creditPurchaseRewards	Internal	Can Modify State	-
_getRepurchaseStatus	Internal	-	-
_repurchaseSingle	Internal	Can Modify State	-
_repurchaseBatch	Internal	Can Modify State	-
_executeRepurchase	Internal	Can Modify State	-

RepurchaserVault			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
_authorizeUpgrade	Internal	Can Modify State	onlyRole
setName	External	Can Modify State	onlyRole

RepurchaserVault			
approve	External	Can Modify State	onlyRole
withdraw	External	Can Modify State	onlyRole

4.3 Vulnerability Summary

[N1] [Medium] Restricted repurchase seizes NFT without payment; canRepurchase view misleads

Category: Design Logic Audit

Content

When the seller's address or the token itself is restricted, `_executeRepurchase` zeroes out the repurchase info, seizes (returns to the pool) or burns the NFT, and then returns immediately -- skipping the payment. The process is irreversible: the repurchase price has already been cleared, so even if the restriction is later lifted the holder can never claim payment again.

At the same time, `_getRepurchaseStatus` uses `payoutBlocked` as a positive disjunct when computing `offerActive`, so a restricted token still shows as "repurchasable" on-chain, luring users into calling `repurchase()` even though the call will never pay out.

Flow: user calls `repurchase()` -> NFT is seized/burned -> 0 USDT received -> repurchase record permanently deleted.

- src/CardPool.sol#L1020-L1033

```
_updateRepurchaseInfo(_tokenId, address(0), 0, 0);   // clears price first
if (isCardBox) {
    ICardBoxNFT(address(nftContract)).burnFrom(_seller, _tokenId);
} else {
    tokenInPool[_tokenId] = true;
    nftContract.safeTransferFrom(_seller, address(this), _tokenId);
}
```

```
if (payoutBlocked) {
    emit RestrictedCardReclaimed(_tokenId, _seller, price);
    return;   // seizure already done, no payment
}

paymentToken.safeTransferFrom(vault, _seller, price); // unreachable when restricted
```

- src/CardPool.sol#L956

```
status.offerActive = termsActive && (status.payoutBlocked || vault != address(0));
// offerActive is also true when payoutBlocked is true, contradicting actual payout
behavior
```

A restricted card holder who calls `repurchase()` has their NFT seized or burned, the repurchase price zeroed out, and receives 0 USDT with no on-chain remedy. The loss equals the full `repurchasePrice`.

Solution

It is recommended to remove or correct the `canRepurchase` view that displays restricted assets as repurchasable, and to fully disclose the consequences of any arrangement that irreversibly withdraws or destroys NFTs without on-chain payment, establish an auditable appeal and compensation mechanism, and if the necessary compliance basis for such an arrangement cannot be demonstrated, refuse repurchase while under restriction and preserve the holder's NFT and claims.

Status

Acknowledged; CardBox Response: Risk accepted; code follow-up required.

[N2] [Low] Repurchase deadline cannot be re-armed during structural lock-out

Category: Design Logic Audit

Content

The repurchase deadline is an absolute timestamp that keeps advancing while the holder has legitimately lost NFT

ownership (e.g. escrowed via `WithdrawCardVault.depositWithFee`, or listed via `CardMarketplace.listCard`). `_executeRepurchase` requires `ownerOf == _seller`, so the holder cannot exercise the repurchase right while the NFT is escrowed.

Once the deadline passes the repurchase right is permanently lost; there is no on-chain mechanism to extend or re-arm it. `setDeadlineInterval` only affects repurchase offers issued in the future.

- src/CardPool.sol#L1003

```
if (nftContract.ownerOf(_tokenId) != _seller) {
    revert CardPoolNotOwner(_tokenId, _seller);
}
```

- src/CardPool.sol#L1008-L1010

```
if (block.timestamp > deadline) {
    revert CardPoolRepurchaseDeadlinePassed(_tokenId, deadline);
}
```

A holder who legitimately escrows or lists the NFT during the deadline window loses the ability to exercise repurchase, and the claim disappears permanently with no on-chain remedy.

Solution

It is recommended to prioritize providing controlled deadline extensions or a suspension mechanism during custody. If the project party continues to use an absolute deadline based on established business rules, it should be prominently indicated on the contract signing and operation interfaces that the countdown will not stop after the NFT leaves the holder's wallet, and market listings are no exception. Early warnings should also be provided for repurchase rights that are approaching expiration but cannot be settled temporarily.

Status

Acknowledged; CardBox Response: Risk accepted.

[N3] [Medium] Unprivileged holder can clear token compliance restriction via repurchase()

Category: Authority Control Vulnerability Audit

Content

`setRestricted(..., true)` requires `OPERATOR_ROLE`, but `repurchase()` is a permissionless entry point whose internal call to `cardManager.setRestricted(_tokenId, TOKEN_RESTRICTION_SLOT, false)` clears the restriction flag. This relies on CardPool's own OPERATOR role on CardManager.

As a result, any user holding a restricted token can clear that token's compliance restriction flag simply by calling `repurchase()`, returning it to circulation in a "clean" state.

- src/CardPool.sol#L966-L970

```
function _repurchaseSingle(uint256 _tokenId, address _seller) internal {
    bool addressRestricted = cardManager.isRestricted(uint256(uint160(_seller)),
ADDRESS_RESTRICTION_SLOT);
    bool tokenRestricted = cardManager.isRestricted(_tokenId, TOKEN_RESTRICTION_SLOT);
    _executeRepurchase(_tokenId, _seller, addressRestricted, tokenRestricted);
    if (tokenRestricted) {
        cardManager.setRestricted(_tokenId, TOKEN_RESTRICTION_SLOT, false);  //
reachable by unprivileged caller
    }
}
```

After clearing the restriction via repurchase, a restricted token re-enters market circulation, bypassing compliance controls.

Solution

It is recommended that token-level restrictions be atomically cleared only after CardPool has successfully retrieved the ordinary NFT or destroyed the CardBox NFT, while retaining the seller address restriction, and that dedicated events and tests be used to prove that holders cannot remove the restrictions while retaining the NFT.

Status

Acknowledged; CardBox Response: Risk accepted.

[N4] [Medium] Privileged roles lack secondary controls, enabling single-key fund extraction

Category: Authority Control Vulnerability Audit

Content

Multiple privileged roles in `CardPool` and `RepurchaserVault` can perform single-step, immediate operations on user funds, without any timelock, multisig, secondary confirmation, or on-chain spending cap:

1. **OPERATOR** can debit any user (USDT / points / voucher) via `purchasePackTrusted` by specifying an arbitrary `_params.user`, without that user's signature. `requestId` only provides idempotency deduplication, not authorization; an empty `_tokenIds` + empty `_cardBoxPrizes` is also a valid call -- funds can be debited from a user's account with nothing given in return. On the EIP-712 signed path, key fields such as repurchase price, repurchase vault, and reward amounts are not bound by the user's signature, so the operator can freely tamper with them.
2. **ADMIN** can instantly change all trust anchors -- `treasury`, `platformRepurchaseVault`, `cardManager`, `deadlineInterval`, `registeredVault` -- with no timelock or two-step confirmation.
3. **WITHDRAWER** can withdraw or approve the entire vault balance via `RepurchaserVault.withdraw` and `approve` without limit, unconstrained by outstanding repurchase obligations.

A single compromised key is sufficient to cause a total loss of funds, with no on-chain delay window for monitoring or governance intervention.

- src/CardPool.sol#L310

```
function purchasePackTrusted(
    PurchasePackTrustedParams calldata _params,
    uint256[] calldata _tokenIds,
    uint256[] calldata _repurchasePrices,
    address[] calldata _repurchaseVaults,
```

```
    CardBoxPrizeInputV2[] calldata _cardBoxPrizes
) external nonReentrant onlyRole(OPERATOR_ROLE) {
    // operator freely specifies _params.user, no user signature required
```

- src/CardPool.sol#L891

```
paymentToken.safeTransferFrom(_user, treasury, _usdtAmount);
// _user is supplied by the operator, no user signature
```

- src/CardPool.sol#L417-L482

```
function setVault(address _vault, bool _enabled) external onlyRole(ADMIN_ROLE) { ... }
function setPlatformRepurchaseVault(address _vault) external onlyRole(ADMIN_ROLE) {
... }
function setCardManager(address _manager) external onlyRole(ADMIN_ROLE) { ... }
function setDeadlineInterval(uint256 _deadlineInterval) external onlyRole(ADMIN_ROLE)
{ ... }
function setTreasury(address _treasury) external onlyRole(ADMIN_ROLE) { ... }
// all trust anchors change instantly, no timelock
```

- src/RepurchaserVault.sol#L96-L114

```
function approve(address token, address spender, uint256 amount)
    external onlyRole(WITHDRAWER_ROLE)
{
    IERC20(token).forceApprove(spender, amount);
    emit RepurchaserVaultApproved(token, spender, amount);
}

function withdraw(address token, address to, uint256 amount)
    external onlyRole(WITHDRAWER_ROLE)
{
    if (to == address(0)) revert RepurchaserVaultZeroAddress();
    IERC20(token).safeTransfer(to, amount);
```

```
        emit RepurchaserVaultWithdrawn(token, to, amount);
    }
```

If any single privileged key is compromised, an attacker can, in one transaction: drain USDT from all authorized users (OPERATOR), redirect all fund flows (ADMIN), or empty the repurchase vault (WITHDRAWER).

Solution

Introduce a timelock or propose-confirm two-step pattern for critical trust anchors. The auditor should also verify that governance and funding roles are held by a Safe multisig that meets reasonable thresholds.

Status

Acknowledged; CardBox Response: Mitigated; legacy paths removed.

[N5] [Information] Effects written after ERC721 callback on five paths (CEI violation)

Category: Unsafe External Call Audit

Content

`_transferNFTs`, `_mintCardBoxPrizes`, and `_transferFromPool` call `nftContract.safeTransferFrom` first (which triggers the recipient's `onERC721Received` callback) before writing state such as `tokenInPool` and `repurchasePrice`. All entry points are protected by `nonReentrant`, blocking same-contract reentrancy, but on-chain state is briefly inconsistent during the callback window.

- src/CardPool.sol#L713-L720

```
nftContract.safeTransferFrom(address(this), _to, tokenId);
tokenInPool[tokenId] = false;                                  // written after transfer

uint256 price = _repurchasePrices[i];
_updateRepurchaseInfo(tokenId, _resolvedVaults[i], price, deadline);
```

No directly exploitable path currently exists. If a future external integrator reads pool state inside the callback, this could evolve into an exploitable issue.

Solution

Move the state writes before the `safeTransferFrom` call, following strict checks-effects-interactions.

Status

Fixed

[N6] [Information] Initialization and deprecated storage hygiene issues

Category: Others

Content

Three independent but related initialization and storage hygiene issues:

1. **`ADMIN_ROLE` is not granted in `initialize`**: `initialize` only grants `DEFAULT_ADMIN_ROLE` and `OPERATOR_ROLE`; 15 ADMIN-gated configuration functions require manual authorization after deployment.
2. **Misleading deprecated slot comment**: the comment on `_deprecatedTotalRepurchaseLiability` still states "Withdrawals may only use funds above this reserve", but the variable is never read or written anywhere, misleading maintainers and auditors.
3. **`RepurchaserVault` permanently locks native ETH**: there is no `receive()` / `fallback()` /payable function, and `withdraw` has no native ETH branch; ETH force-sent via `selfdestruct` or as a coinbase reward is unrecoverable.

- src/CardPool.sol#L176-L178

```
_grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
_grantRole(OPERATOR_ROLE, msg.sender);
// ADMIN_ROLE is missing
```

- src/CardPool.sol#L128-L131

```
/// @dev Appended for upgrade safety. Withdrawals may only use funds above this
reserve.
```

```
uint256 public _deprecatedTotalRepurchaseLiability;  // never read or written
```

- src/RepurchaserVault.sol#L108-L114

```
function withdraw(address token, address to, uint256 amount)
    external onlyRole(WITHDRAWER_ROLE)
{
    if (to == address(0)) revert RepurchaserVaultZeroAddress();
    IERC20(token).safeTransfer(to, amount);   // ERC20 only, no native ETH branch
    emit RepurchaserVaultWithdrawn(token, to, amount);
}
```

No direct exploitable risk. The missing `ADMIN_ROLE` grant at deployment has already been resolved off-chain; the deprecated slot comment may mislead maintenance decisions; ETH lock-up only occurs in extreme, non-standard scenarios.

Solution

It is recommended to grant `ADMIN_ROLE` in `initialize` and confirm via storage layout tests that the five deprecated slots remain unchanged before and after the upgrade. For the residual risk that native ETH forcibly transferred in cannot be retrieved, a governance-controlled recovery entry should be added, or—after explicitly disclosing and accepting the risk—a temporary upgrade recovery plan approved by the Safe multisig should be prepared in advance.

Status

Acknowledged; Fixed in part; CardBox Response: Residual risk accepted.

[N7] [Suggestion] Repurchase payout has no solvency guarantee

Category: Design Logic Audit

Content

Repurchase obligations are written by three paths -- `_transferFromPool`, `_transferNFTs`, `_mintCardBoxPrizes` -- but at settlement `_executeRepurchase` calls

`paymentToken.safeTransferFrom(vault, seller, price)` directly without checking the vault's balance or allowance.

`RepurchaserVault.withdraw` can freely drain the balance. In addition, settlement never re-checks `registeredVault` status -- after an operator deregisters a vault via `setVault(V, false)`, tokens still bound to that vault continue to be debited from it normally.

- src/RepurchaserVault.sol#L108-L114

```
function withdraw(address token, address to, uint256 amount)
    external onlyRole(WITHDRAWER_ROLE)
{
    if (to == address(0)) revert RepurchaserVaultZeroAddress();
    IERC20(token).safeTransfer(to, amount);  // no reserve kept
    emit RepurchaserVaultWithdrawn(token, to, amount);
}
```

- src/CardPool.sol#L1013-L1016

```
vault = repurchaseVault[_tokenId];
if (vault == address(0)) vault = platformRepurchaseVault;
if (vault == address(0)) revert CardPoolRepurchaseVaultNotSet();
// missing registeredVault[vault] check and balance/allowance check
```

When the vault has insufficient funds, `repurchase()` keeps reverting until the deadline expires, permanently forfeiting the user's claim right. Tokens bound to a deregistered vault continue to be debited from it.

Solution

It is recommended to prioritize ensuring repurchase solvency through on-chain liability accounting and reserve constraints. If the project side opts for operational control, existing obligations should remain bound to the original Vault, with continuous monitoring of balances and authorizations, clearly defined fund replenishment thresholds, and an abnormal handling process established before maturity. Once the Vault is deregistered, it must not undertake any

new obligations.

Status

Acknowledged; CardBox Response: Risk accepted; operationally controlled.

[N8] [Suggestion] EIP-712 signature authorization incomplete

Category: Authority Control Vulnerability Audit

Content

`PURCHASE_PACK_TYPEHASH` only signs 9 fields (`user, packId, quantity, paymentMethod, usdtAmount, voucherAmount, pointsAmount, nonce, deadline`); the following key parameters are not bound by the user's signature:

- `cardsHash`: the check in `_verifyPurchaseSignature` compares two values both supplied by the same caller -- a self-consistency check, not a signed commitment.
- `returnPoints`, `voucherPrize`: reward amounts are freely set by the operator.
- `_repurchasePrices`, `_repurchaseVaults`: repurchase terms are entirely unconstrained.
- No minimum payment amount: `_validatePaymentAmounts` does not require the amount to be `> 0`; `safeTransferFrom(user, treasury, 0)` executes legally without transferring any asset.
- src/libraries/Constants.sol#L30-L33

```
bytes32 constant PURCHASE_PACK_TYPEHASH = keccak256(
    "PurchasePack(address user,uint32 packId,uint16 quantity,uint8
paymentMethod,uint256 usdtAmount,uint256 voucherAmount,uint256 pointsAmount,uint256
nonce,uint256 deadline)"
);
```

- src/CardPool.sol#L907-L928

```
function _validatePaymentAmounts(
    PaymentMethod _method,
    uint256 _usdtAmount,
```

```
    uint256 _voucherAmount,
    uint256 _pointsAmount
) internal pure {
    bool valid;
    if (_method == PaymentMethod.USDT) {
        valid = _voucherAmount == 0 && _pointsAmount == 0;
    } else if (_method == PaymentMethod.Voucher) {
        valid = _usdtAmount == 0 && _pointsAmount == 0;
    } else if (_method == PaymentMethod.Points) {
        valid = _usdtAmount == 0 && _voucherAmount == 0;
    } else if (_method == PaymentMethod.UsdtVoucher) {
        valid = _pointsAmount == 0;
    } else {
        revert CardPoolInvalidPaymentMethod();
    }
    // no check that the total amount is > 0
    if (!valid) {
        revert CardPoolInvalidPaymentAmounts(uint8(_method), _usdtAmount,
_voucherAmount, _pointsAmount);
    }
}
```

OPERATOR can tamper with draw results, repurchase price, repurchase vault, and reward amounts, and construct zero-cost purchases. Risk is low under the trusted-operator assumption, but escalates to Critical if the operator key is compromised.

Solution

It is recommended that, while retaining V2's dual authorization and legitimate zero-amount or empty-result operations, an independent commitment or off-chain tamper-proof verification be added to settlement.

Status

Acknowledged; CardBox Response: Mitigated; residual risk accepted.

[N9] [Suggestion] cardBoxTemplateIdByToken not reset after repurchase settlement

Category: Design Logic Audit

Content

`cardBoxTemplateIdByToken` is written in `_mintCardBoxPrizes`, but `_updateRepurchaseInfo` only clears the price, vault, and deadline -- it never clears `cardBoxTemplateIdByToken`. `_executeRepurchase` determines whether a token is a CardBox via `cardBoxTemplateIdByToken[_tokenId] > 0`.

If a burned token ID is later re-minted by the NFT contract and deposited back into the pool, `_depositCard` has no guard requiring `cardBoxTemplateIdByToken == 0`, so the token would be misclassified as a CardBox and routed down the burn branch.

- src/CardPool.sol#L999

```
uint256 templateId = cardBoxTemplateIdByToken[_tokenId];
bool isCardBox = templateId > 0;  // stale value causes misclassification
```

The stale `cardBoxTemplateIdByToken` value causes a re-deposited token to take the wrong branch, leaving the pool's bookkeeping inconsistent.

Solution

Clear `cardBoxTemplateIdByToken[_tokenId]` in the `isCardBox` branch of `_executeRepurchase`.

Status

Fixed

[N10] [Suggestion] Non-upgradeable ReentrancyGuard inherited by proxied contract

Category: Others

Content

`CardPool` inherits the non-upgradeable `@openzeppelin/contracts/utils/ReentrancyGuard` instead of the upgradeable variant. Its `_status` variable occupies a fixed storage slot that is never initialized in `initialize`. It

works correctly under the current OZ v5 layout, but this slot affects the storage layout and must be manually preserved on any future upgrade.

- src/CardPool.sol#L10

```
import {ReentrancyGuard} from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
```

No direct vulnerability currently. A future upgrade that fails to preserve the storage layout could break reentrancy protection or cause a storage collision.

Solution

It is recommended to retain the existing `ReentrancyGuard` for deployed proxies only if storage layout checks and upgrade regression tests prove it safe, and to evaluate using `ReentrancyGuardTransient` in new deployments that support EIP-1153, so as to avoid introducing higher upgrade risk by replacing the guard implementation.

Status

Acknowledged; CardBox Response: Risk accepted; existing guard preserved.

[N11] [Suggestion] DOMAIN_SEPARATOR cached at initialize, stale after chain-id change

Category: Replay Vulnerability

Content

`DOMAIN_SEPARATOR` caches `block.chainid` at `initialize` time and is never recomputed afterward. After a chain fork, `chainid` changes, but the contract keeps using the old value, so old signatures remain valid on the forked chain.

- src/CardPool.sol#L180-L189

```
DOMAIN_SEPARATOR = keccak256(
    abi.encode(
        keccak256("EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)"),
        keccak256(bytes("CardBox Pool")),
```

```
            keccak256(bytes("1")),
            block.chainid,
            address(this)
        )
    );
```

After a chain fork, old signatures can be replayed on the forked chain. Exploitation requires an OPERATOR to submit the replayed signature.

Solution

Adopt the OZ `EIP712` pattern that recomputes the domain separator when `block.chainid != cachedChainId`.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002608050001	SlowMist Security Team	2026.08.04 - 2026.08.05	Medium Risk

Summary conclusion: The SlowMist security team use a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 3 medium risk, 1 low risk, 5 suggestion vulnerabilities.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist