LoL Land

Audit Report

2nd June 2025

FAILSAFE

Table of Contents

Executive Summary

LoL Land engaged the FailSafe security team to perform a focused security review of its core contracts; LLManager and LLNFTUpgradeable, deployed on Abstract Chain, an EVM-compatible chain.

Through rigorous manual code inspection and targeted Foundry PoCs, we uncovered vulnerabilities in token handling, signature domain separation, and parameter validation. These issues, although of centralized nature with medium-impact, could be exploited to misalign on-chain accounting, enable cross-instance fund drains, and bypass business-logic constraints.

Recommendations include adopting SafeERC20, EIP-712 domain tagging, and strict on-chain bounds checks.

Project Details

Project	LOL Land - Abstract Chain (EVM-Compatible)
URL	https://www.lol.land
Source Code	https://github.com/YGG-Vietnam/LOLLANDSC
Initial Commit	57f125c79513f55ebc0e7871036124a80d338bf9
Timeline	Initial Report - 19th May 2025 - 21st May 2025
	Final Report - 21st May 2025 - 2nd June 2025

Structure & Organization of Audit Report

Issues are tagged as "Open", "Acknowledged", "Partially Resolved", "Resolved" or "Closed" depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the Smart Contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the Smart Contract to compile or operate in a significant way.
Medium	The issue affects the ability of the Smart Contract to operate in a way that doesn't significantly hinder its behavior.
Low	The issue has minimal impact on the Smart Contract's ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the Smart Contract's operation.

Audit Approach

The following are areas of concern will be investigated during the audit, along with any similar potential issues:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the Smart Contracts;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution of the Smart Contracts;
- Vulnerabilities in the Smart Contracts code;
- Protection against malicious attacks and other ways to exploit Smart Contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

The following schedule will be followed:

- Code review completed and delivery of Initial Audit Report
- Client responds with fixes and/or acknowledgments for all findings
- Security team validates the fixes and/or acknowledgments
- Verification completed
- Delivery of Final Audit Report

Project Goals

1. Can both LLManager and LLNFTUpgradeable initialize and upgrade without leaving any gaps in access control or uninitialized modules?
2. Do the deposit, claim, and withdrawal functions always maintain accurate on-chain balances and fail cleanly on any mismatch?
3. Are off-chain signatures tightly bound to a single contract instance, chain ID, nonce, and expiration to prevent any replay or cross-instance reuse?
4. Can owner-only operations—pausing, unpausing, signer/backend updates, emergency withdrawals—only be executed by the rightful owner and emit clear audit events?
5. Is the NFT lifecycle (minting, metadata setting, burning, royalties) fully compliant with ERC-721 and ERC-2981 standards, without hidden edge cases?
6. Are all external calls, especially ERC-20 transfers and token claims, protected by SafeERC20 methods and nonReentrant guards to eliminate reentrancy or silent-failure risks?
7. Are every user-supplied parameter—amounts, package IDs, avatar arrays—validated with strict ranges and uniqueness checks to prevent overload, DOS, or logic corruption?
8. Does the overall implementation follow upgradeable-contract best practices and maintain gas efficiency without unnecessary complexity?

Audit Methodology

FailSafe employs a multi-layered approach to Smart Contract security audits:

Threat Modelling: We identify critical assets, enumerate potential threats, assess vulnerabilities, and prioritize risks based on severity and impact.

Manual Code Review: Our experts conduct a detailed, line-by-line review of the code, analyzing business logic, access controls, gas efficiency, and external dependencies.

Functional Testing: Using frameworks like Hardhat and Foundry, we perform comprehensive functional and integration tests to ensure correct and secure Smart Contract behavior.

Fuzzing & Invariant Testing: Advanced techniques such as fuzzing and invariant testing are used to uncover hidden vulnerabilities and verify Smart Contract consistency under diverse scenarios.

Edge Case Analysis: We rigorously test for extreme inputs, exception handling, concurrency, and non-standard scenarios to ensure robust Smart Contract performance.

Reporting & Recommendations: Our reports clearly describe each issue, its impact, location, root cause, and provide actionable remediation steps and best practice guidelines.

Remediation Support: We work closely with your team to implement and validate fixes, followed by a final assessment to confirm all issues are resolved.

FailSafe's process ensures your Smart Contracts are secure from initial deployment through ongoing operation, providing proactive and comprehensive protection.

In-scope Files

All Solidity contracts under contracts/, including:

- contracts/LLManager.sol
- contracts/LLNFTUpgradeable.sol

Out of Scope

- Off-chain backend or UI logic
- Dependencies and supporting Libraries

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved	Closed
Critical	-	-	-	-	-	-
High	2	-	-	1	1	-
Medium	2	-	1	-	1	-
Low	1	-	1	-	-	-
Info	1	-	1	-	-	-
Total	6	0	3	1	2	0

#	Findings	Severity	Status
1	Missing Domain Separation Enables Cross-Instance Signature Replay	High	Resolved
2	Zero-Amount ETH Deposits Enable Package-ID Spam And Backend Griefing	High	Partially Resolved
3	Unsafe ERC-20 Deposit Calls Allow Silent Failures	Medium	Resolved
4	Unrestricted redeemType Parameter Allows Arbitrary Values	Medium	Acknowledged
5	No Maximum Cap on Per-Avatar Mint Allowances	Low	Acknowledged
6	Best-Practice Suggestions	Info	Acknowledged

Finding 1: Missing Domain Separation Enables Cross-Instance Signature Replay

Severity: High

Status: Resolved

Source: LLManager.sol (claimTokens(...))

Description:

`claimTokens` verifies a backend-generated ECDSA signature over abi.encodePacked(user, amount, nonce, deadline, backendId, redeemType). However, it omits any binding to the specific contract instance (e.g. `address(this)`) or chain identifier (`chainID`). This allows a valid `signature` for one `proxy` instance to be replayed on an entirely separate `proxy` deployment that shares the same signer and token, or an entirely different chain.

Impact:

Cross-deployment theft: A single signed claim can be used to drain funds from N different instances of LLManager deployed on the same or forked chains. Undermined separation: Backends cannot safely issue signatures per-instance, as any compromise or replay affects all deployments sharing the signer key. Economic loss: Attackers multiply claims without additional signatures, draining vaults across environments.

Code:

```
function claimTokens(...) {
    uint256 nonce = userNonces[msg.sender]++;
    bytes32 hash = keccak256(
        abi.encodePacked(
            msg.sender,
            _amount,
            nonce,
            _deadline,
            _backendId,
            _redeemType
        )
    );
    bytes32 ethSignedHash = hash.toEthSignedMessageHash();
}
```

Proof of Concept:

`test_AllowsSignatureReplayAcrossContractsFAIL` deploys two proxies (`managerA`, `managerB`) with identical `signer` and `token`. A single `signature` is consumed by `managerA.claimTokens(...)` then reused to successfully extract the same `amount` from `managerB`.

```
function test_AllowsSignatureReplayAcrossContractsFAIL() public {
    // Deploy a second LLManager with the same signer and token
    LLManager impl2 = new LLManager();
```

```
    bytes memory init2 = abi.encodeWithSelector(
        LLManager.initialize.selector,
        signerEoa,
        address(yggToken)
    );
    TransparentUpgradeableProxy p2 = new TransparentUpgradeableProxy(
        address(impl2),
        address(proxyAdmin),
        init2
    );
    LLManager manager2 = LLManager(address(p2));

    // Fund both managers with 20 tokens
    uint256 fundAmount = 20 ether;
    yggToken.transfer(address(manager), fundAmount);
    yggToken.transfer(address(manager2), fundAmount);
    assertEq(yggToken.balanceOf(address(manager)), fundAmount);
    assertEq(yggToken.balanceOf(address(manager2)), fundAmount);

    // Build a single signature for a 10-token claim
    uint256 amount = 10 ether;
    uint256 deadline = block.timestamp + 1 days;
    uint256 backendId = 1;
    uint256 redeemType = 2;
    uint256 nonce = manager.userNonces(address(this)); // both start at 0
    bytes32 h = keccak256(
        abi.encodePacked(
            address(this),
            amount,
            nonce,
            deadline,
            backendId,
            redeemType
        )
    );
    bytes32 ethH = ECDSAUpgradeable.toEthSignedMessageHash(h);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(privateKey, ethH);
    bytes memory sig = abi.encodePacked(r, s, v);

    // Claim on manager1 then replay on manager2
    manager.claimTokens(amount, deadline, backendId, redeemType, sig);
    manager2.claimTokens(amount, deadline, backendId, redeemType, sig);

    // Both managers lost 10 tokens each - signature replay succeeds
    assertEq(yggToken.balanceOf(address(manager)), fundAmount - amount);
    assertEq(yggToken.balanceOf(address(manager2)), fundAmount - amount);
}
```

Remediation:

Consider migrating to EIP-712 structured signing to formalize domain data and prevent all replay variants or, integrate explicit domain separation into the signed payload. For example:

```
bytes32 hash = keccak256(
  abi.encode(
```

```
        address(this),          // contract address
        block.chainid,          // chain ID
        msg.sender,
        amount,
        nonce,
        deadline,
        backendId,
        redeemType
    )
);
```

Finding 2: Zero-Amount ETH Deposits Enable Package-ID Spam And Backend Griefing

Severity: High

Status: Partially Resolved

Source: LLNFTUpgradeable.sol (depositToken(address _token, uint256 _amount, uint256 _packageId))

Description:

The `depositToken` function is used to record on-chain deposits of `ETH` or whitelisted ERC-20 `tokens`, emitting a `TokenDeposited(user, token, amount, packageId)` event for each call. In the `ETH` branch `(_token == address(0))`, it only checks that `msg.value == _amount`, so calling `depositToken(0x0, 0, anyPackageId)` with `zero ETH` always succeeds, since both sides of the equality are `0`. There is no requirement that `_amount` be non-zero, nor any check that a given `packageId` hasn't already been used by the same user or used at all.

In most NFT or gaming flows, `packageId` ties an on-chain deposit to a specific off-chain order, bundle, or in-game purchase. Back-end services typically listen for `TokenDeposited` events, match each unique `packageId` to a user's purchase record or a seasonal event trackables (e.g. 201, 4401, 509, 3099), and then grant assets or update game state. When attackers can emit thousands of zero-value deposit events with arbitrary or sequential packageId values at almost no cost, they can: Flood the Event Stream (Denial-of-Service): Cheap zero-value transactions spam the deposit logs, making it hard for your indexer and back-end to spot real deposits among the noise. Front-Run Legitimate Deposits: By submitting `depositToken(0, 0, 42)` just before a real user's purchase with `packageId = 42`, the back-end sees the event twice, or in the wrong order, and may ignore or misassign the real purchase. Duplicate or Steal IDs: Replaying the same `packageId` can trick anti-fraud systems into believing one deposit happened multiple times, potentially suspending honest users or triggering false alerts.

Impact:

Broken Purchase Flows: Genuine buyers may find their orders unrecognized or credited to the wrong account. Operational Overhead: Teams must build complex filters or manual reviews to weed out zero-amount spam. User Distrust: Players hit with "missing" assets or false fraud flags will lose faith in the platform.

Code:

```
/**
 * @dev Deposit tokens with a unique package ID for backend tracking
 * @param _token Address of token being deposited (address(0) for ETH)
 * @param _amount Amount of tokens to deposit
```

```
 * @param _packageId Unique identifier for this package purchase
 */
function depositToken(
    address _token,
    uint256 _amount,
    uint256 _packageId
) external payable {
    if (!whitelistedTokens[_token]) revert InvalidToken();
    if (_token == address(0)) {
        if (msg.value != _amount) revert InvalidAmount();
    } else if (msg.value != 0) revert InvalidAmount();

    userDeposits[msg.sender][_token] += _amount;
    totalTokenBalances[_token] += _amount;

    if (_token != address(0))
        IERC20Upgradeable(_token).transferFrom(
            msg.sender,
            address(this),
            _amount
        );

    emit TokenDeposited(msg.sender, _token, _amount, _packageId);
}
```

Remediation:

Disallow zero-amount deposits to ensure every event represents real value:

```
require(_amount > 0, "Deposit amount must be > 0");
```

Track and forbid duplicate `packageId` per user so each deposit can only occur once:

```
mapping(address => mapping(uint256 => bool)) usedPackageIds;
require(!usedPackageIds[msg.sender][_packageId], "Duplicate packageId");
usedPackageIds[msg.sender][_packageId] = true;
```

(Optional) Use on-chain sequencing, e.g., an auto-increment counter or a preapproved registry, to guarantee valid `packageId`s and preserve order.

Auditor Response:

A new change has rendered withdrawCollectedFunds current implementation meaningless, since deposits now relay funds to the treasury wallet, the getAvailableFunds() will usually return 0 (except accidental ETHs recieved).

Ultimately, the withdrawCollectedFunds() will not be able to return any funds to the owner since address(this) does not hold any user funds in lieu of the treasury wallet.

Finding 3: Unsafe ERC-20 Deposit Calls Allow Silent Failures

Severity: Medium

Status: Resolved

Source: LLManager.sol (depositToken(address _token, uint256 _amount, uint256 _bundleId))

Description:

Both contracts accept arbitrary ERC-20 tokens via raw `transferFrom/transfer` calls without checking `return` values. Many tokens, e.g. non-standard ones like ERC777 signal failure by returning `false` instead of reverting. In this pattern, a failed transfer silently passes through, yet the contract still:

Updates internal mappings (e.g. `userDeposits` or emits `TokenDeposited`). Leaves the token `balance` unchanged (no tokens received).

This mismatch between on-chain accounting and actual token movement corrupts `userDeposits` or bundle tracking, leading to inconsistent state and potential user confusion or loss of funds.

Impact:

Ghost deposits: Users see "successful" deposit events but the tokens never arrive. Denial of service: Legitimate users may lose funds or be unable to mint/redeem because the contract's accounting is out of sync. Attack vector: Malicious tokens can be whitelisted and used to sabotage project operations by consistently returning `false` in `transferFrom`, spamming the deposit flow without delivering any assets.

Code:

```
function depositToken(
    address _token,
    uint256 _amount,
    uint256 _packageId
) external payable {
    if (!whitelistedTokens[_token]) revert InvalidToken();
    if (_token == address(0)) {
        if (msg.value != _amount) revert InvalidAmount();
    } else if (msg.value != 0) revert InvalidAmount();

    userDeposits[msg.sender][_token] += _amount;
    totalTokenBalances[_token] += _amount;

    if (_token != address(0))
        IERC20Upgradeable(_token).transferFrom(
            msg.sender,
            address(this),
            _amount
```

```
        );

    emit TokenDeposited(msg.sender, _token, _amount, _packageId);
}
```

Proof of Concept:

The testcase `test_AllowsSilentlyFailingUnsafeERC20DepositFAIL` deploys an ERC-20 stub that always returns `false` on `transferFrom`. The test calls the `depositToken` method, sees the event emitted, but confirms the contract's token `balance` remains `zero` and the user's balance remains intact.

```
function test_AllowsSilentlyFailingUnsafeERC20DepositFAIL() public {
    // Deploy a token whose transferFrom returns false
    MaliciousERC20 mf = new MaliciousERC20();
    mf.mint(address(this), 100 ether);

    uint256 initialBalance = mf.balanceOf(address(this));

    // Whitelist it in the manager
    manager.setWhitelistedToken(address(mf), true);

    // Pre-condition: manager has zero balance
    assertEq(mf.balanceOf(address(manager)), 0);

    // Approve the manager to transfer tokens
    mf.approve(address(manager), 10 ether);

    // Call depositToken; under raw ERC20 this will not revert
    manager.depositToken(address(mf), 10 ether, 1);

    // Post-condition: manager still holds zero tokens
    assertEq(mf.balanceOf(address(manager)), 0);
    // User still retains full balance
    uint256 finalBalance = mf.balanceOf(address(this));
    assertEq(finalBalance - initialBalance, 0);
}
```

Remediation:

Replace raw IERC20.transferFrom/transfer calls with OpenZeppelin's SafeERC20Upgradeable.safeTransferFro and safeTransfer. Revert on any failed transfer to guarantee balance consistency:

```
using SafeERC20Upgradeable for IERC20Upgradeable;

IERC20Upgradeable(_token).safeTransferFrom(msg.sender, address(this), _amount);
```

Finding 4: Unrestricted redeemType Parameter Allows Arbitrary Values

Severity: Medium

Status: Acknowledged

Source: LLManager.sol (claimTokens(...))

Description:

The `_redeemType` argument is used to categorize different redemption flows but is never validated on-chain. An attacker can set `redeemType` to any uint256, including values outside the intended enumeration, without triggering a revert.

Impact:

Workflow confusion: Off-chain systems expecting only a small set of `redeemType` values (e.g. `0, 1, 2`) may log or process "unknown" types incorrectly. Privilege escalation: If certain types map to higher-value rewards, an attacker could request an unanticipated `redeemType` to gain unauthorized benefits. Auditing blind spots: Unchecked types can slip under the radar of monitoring tools, enabling stealthy fraud.

Code:

```
function claimTokens(
    uint256 _amount,
    uint256 _deadline,
    uint256 _backendId,
    uint256 _redeemType,
    bytes calldata _signature
) external whenNotPaused {
    require(block.timestamp <= _deadline, "Signature expired");
    require(_amount > 0, "Amount must be greater than 0");
    require(
        yggToken.balanceOf(address(this)) >= _amount,
        "Insufficient contract balance"
    );

    uint256 nonce = userNonces[msg.sender]++;
    bytes32 hash = keccak256(
        abi.encodePacked(
            msg.sender,
            _amount,
            nonce,
            _deadline,
            _backendId,
            _redeemType
        )
    );
    bytes32 ethSignedHash = hash.toEthSignedMessageHash();
}
```

Proof of Concept:

`test_acceptsOutOfRangeRedeemTypeFAIL` uses `redeemType = type(uint256).max` in a valid `signature`. The contract accepts and processes the claim, demonstrating zero bounds checking.

```
function test_acceptsOutOfRangeRedeemTypeFAIL() public {
    // Fund the manager with YGG tokens
    uint256 fundAmount = 50 ether;
    yggToken.transfer(address(manager), fundAmount);

    // Record starting nonce and balance
    uint256 startNonce = manager.userNonces(address(this));
    uint256 startBal = yggToken.balanceOf(address(this));

    // Prepare parameters with an absurdly large redeemType
    uint256 amount = 5 ether;
    uint256 deadline = block.timestamp + 1 days;
    uint256 backendId = 0;
    uint256 redeemType = type(uint256).max;

    // Sign the payload
    bytes32 hash = keccak256(
        abi.encodePacked(
            address(this),
            amount,
            startNonce,
            deadline,
            backendId,
            redeemType
        )
    );
    bytes32 ethHash = ECDSAUpgradeable.toEthSignedMessageHash(hash);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(privateKey, ethHash);
    bytes memory sig = abi.encodePacked(r, s, v);

    // Act: claimTokens with out-of-range redeemType
    manager.claimTokens(amount, deadline, backendId, redeemType, sig);

    // Assert: succeeds, balance increased and nonce incremented
    assertEq(manager.userNonces(address(this)), startNonce + 1);
    assertEq(yggToken.balanceOf(address(this)) - startBal, amount);
}
```

Remediation:

If using an enum off-chain, mirror it on-chain and revert on out-of-range values. Additionally, enforce an explicit whitelist or max value check:

```
require(_redeemType <= MAX_VALID_REDEEM_TYPE, "Invalid redeemType");
```

Finding 5: No Maximum Cap on Per-Avatar Mint Allowances

Severity: Low

Status: Acknowledged

Source: LLNFTUpgradeable.sol (purchasePackageWithDeposit(...))

Description:

When granting `mint allowances` via `purchasePackageWithDeposit`, the contract increments `userAvatarMints[user][avatarHash]` by the received `mint count`, but imposes no upper limit. A malicious backend key can issue a single signature allowing millions of mints, or a bug could cause runaway increments.

Impact:

Scarcity collapse: Unlimited free mints destroy NFT rarity and devalue the collection. Revenue loss: Attackers can mint unlimited NFTs at no cost, bypassing intended payment flows. Downstream denial-of-service: Marketplace or indexing infrastructure may break under an unexpectedly massive mint flood.

Code:

```
function purchasePackageWithDeposit(
    address _user,
    address _token,
    uint256 _amount,
    string[] calldata _avatarIds,
    uint256[] calldata _mintCounts,
    bytes memory _signature
) external {
    if (_avatarIds.length != _mintCounts.length) revert ArraysLengthMismatch();
    if (userDeposits[_user][_token] < _amount) revert InsufficientDeposit();

    _validateSignature(_user, _token, _amount, _avatarIds, _mintCounts,
        _signature);
    userDeposits[_user][_token] -= _amount;
    totalTokenBalances[_token] -= _amount;

    // Update mint allowances
    for(uint256 i = 0; i < _avatarIds.length; i++) {
        bytes32 avatarHash = keccak256(abi.encodePacked(_avatarIds[i]));
        userAvatarMints[_user][avatarHash] += _mintCounts[i];
        emit MintAllowanceUpdated(_user, _avatarIds[i], userAvatarMints[_user][
            avatarHash]);
    }
    emit DepositUsed(_user, _token, _amount, userDeposits[_user][_token]);
}
```

Remediation:

Validate `_mintCounts[i]` against a per-avatar maximum before updating. Alternatively, impose a reasonable cap per user and per avatar, for example:

```
uint256 newAllowance = userAvatarMints[_user][avatarHash] + _mintCounts[i];
require(newAllowance <= MAX_MINTS_PER_AVATAR, "Mint cap exceeded");
userAvatarMints[_user][avatarHash] = newAllowance;
```

Finding 6: Best-Practice Suggestions

Severity: Info

Status: Acknowledged

Source: LLNFTUpgradeable.sol and LLManager.sol

Description:

Several best practices and improvements were identified that, while not introducing new high-severity risks on their own, can significantly strengthen robustness, observability, and user experience:

Dedicated Event & Balance Verification in `withdrawYgg`: - Emit a YggWithdrawn(address indexed owner, uint256 amount) event.

Sanity Checks & Event in `withdrawCollectedFunds`: - Before computing the on-chain balance as `balance = address(this).balance - totalTokenBalances[_token]`, place an assertion or a constraint to ensure the underflow is gracefully handled: `address(this).balance >= totalTokenBalances[_token]` - Emit `CollectedFundsWithdrawn(address indexed owner, address indexed token, uint256 amount)` event for transparent fund flows.

Batch-Size Limit in `purchasePackageWithDeposit`: - Enforce a max cap for the incoming array as a limit on `_avatarIds` (e.g. <= 50) to prevent accidental out-of-gas when processing large arrays signed by the backend.

Remediation:

Adopt the above checks and events as low-overhead, high-value enhancements.

Tag each with require or event emission in the relevant function to improve on-chain diagnostics and off-chain integration.

Disclaimer

This audit report (“Report”) is provided by FailSafe (“Auditor”) for the exclusive use of the client (“Client”). The audit scope is limited to a technical review of the Smart Contract code supplied by the Client.

While FailSafe has made every effort to identify vulnerabilities and deviations from best practices, we do not guarantee the absence of all security issues or that the Smart Contract will function as intended in every environment.

FailSafe is not liable for any losses or damages arising from the use, misuse, or inability to use the Smart Contract. This Report is not an endorsement or certification of the Smart Contract and may not be shared or reproduced without FailSafe’s written consent.

The Client is solely responsible for the deployment, operation, and further testing of the Smart Contract, as well as for implementing any recommended changes. FailSafe may update this Report if new information becomes available, and the Client is encouraged to maintain ongoing security reviews.

By using this Report, the Client accepts these terms and conditions.

Appendix

Test Cases Executed:

The full suite of 11 Foundry tests exercised both core "happy-path" behaviors and some of the FailSafe team's security edge cases designed for targeted exploit scenarios.

We began by verifying that both LLManager and LLNFTUpgradeable initialize correctly and respect pause/unpause semantics. Next, we confirmed that claimTokens resists reentrancy and that failed signatures do not inadvertently burn nonces. We also ensured ERC-2981 support is intact and that the purchase-package flow correctly enforces a per-user nonce.

The three "FAIL" tests; unsafe ERC-20 deposits, cross-contract signature replay, and out-of-range redeemType, all passed, demonstrating that those vulnerabilities exist as expected.

In total, all 11 tests passed with zero failures, validating both the presence of the three identified issues and the correct operation of the remaining functionality.

```
Ran 11 tests for test/LLUpgrade.t.sol:LLUpgradeTest
[PASS] test_AllowsSignatureReplayAcrossContractsFAIL() (gas: 1616214)
[PASS] test_AllowsSilentlyFailingUnsafeERC20DepositFAIL() (gas: 752813)
[PASS] test_ManagerInitializationPASS() (gas: 32187)
[PASS] test_NFTInitializationPASS() (gas: 31565)
[PASS] test_PausingUninitializeFunctionalityPASS() (gas: 38996)
[PASS] test_StoppingReentrancyInClaimTokensPASS() (gas: 2248240)
[PASS] test_StopsNonceBurnOnFailedSignaturePASS() (gas: 91543)
[PASS] test_acceptsOutOfRangeRedeemTypeFAIL() (gas: 99627)
[PASS] test_missingERC2981InitPASS() (gas: 41845)
[PASS] test_reentrancyDepositTokenPASS() (gas: 807667)
[PASS] test_signatureReplayPurchasePackagePASS() (gas: 223342)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 2.90ms (7.35ms CPU time)
```