ZEALYNX SECURITY

Web3 Security & Smart Contract Development

Golden Grid

Smart Contract Audit

November 17, 2025

Prepared by Zealynx
contact@zealynx.io

Carlos (Bloqarl) @TheBlockChainer

Contents

1. About Zealynx

Zealynx, founded in January 2024 by Carlos (Bloqarl), specializes in smart contract audits, and development. Our services include comprehensive smart contract audits, application security audits, such as pentesting, and AI Audits. We are trusted by clients such as Mangrove, Ample protocol, Lido, Inverter, Matchain, and Paragon.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website Zealynx.io and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

A time-boxed independent Solidity smart contract Audit of the Golden Grid protocol was conducted by Zealynx, with a focus on the potential security vulnerabilities.

We performed the security evaluation based on the agreed scope during 2 weeks, following our systematic approach and methodology. Based on the scope and our performed activities, our security assessment revealed 1 Critical, 3 High, 1 Medium, 4 Low and 1 Informational security issues.

3.2 About Golden Grid

Golden Grid is a decentralized lottery protocol built on blockchain technology that combines pixel-based gameplay with cryptographically secure randomness. The system features a grid of 98,280 unique pixels that players can purchase. Each pixel number is mathematically converted into a distinct 5-bit combination using lexicographic enumeration across 28 possible bit positions.

The protocol utilizes Chainlink VRF (Verifiable Random Function) to ensure provably fair lottery draws. Winners are determined by calculating bit overlaps between randomly drawn pixels and player-owned pixels using efficient bitwise operations. The system includes sophisticated financial distribution mechanisms for multi-shareholder reward allocation and implements advanced mathematical techniques like rejection sampling to eliminate bias in random number generation.

3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity language and includes ~770 nSLOC - normalized source lines of code. The codebase consists of multiple Solidity smart contracts implementing combinatorial mathematics libraries, VRF callback handlers, and financial accounting systems with precision arithmetic for reward calculations and shareholder distributions.

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing**, and **meticulous manual security review.**

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality**: We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices**: Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments**: We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Over a 2-week engagement, Zealynx team conducted a security review of the Golden Grid Smart Contract implementation.

The audit focused on identifying vulnerabilities, logic flaws, and implementation-level issues that could affect the lottery protocol's random number generation, pixel-to-bitmap conversions, financial distribution mechanisms, and overall system fairness.

A total of 10 issues were identified and categorized as follows:
- 1 Critical severity
- 3 High severity
- 1 Medium severity
- 4 Low severity
- 1 Informational

The initial commit hash audited is:
- dae05f0a1746d81bf3bc97b461af40310aa03ca3

The key findings include:

- **VRF callback failure deadlock:** The protocol lacks recovery for failed VRF callbacks, causing permanent deadlock where isPending stays locked. If callbacks revert, all future draws become impossible and user funds are permanently locked.
- **Biased lottery distribution:** The transformRandomToPixel function executes identical modulo operations in both branches, negating bias correction. Pixel numbers 0-57,895 have ~0.059% higher winning probability, compromising lottery fairness.
- **Systematic accounting corruption:** The totalUnclaimedProceeds mechanism tracks only new rewards but subtracts complete liabilities during withdrawals. This causes systematic corruption, enabling over-allocation and preventing legitimate withdrawals.
- **Fund loss during zero-shareholder periods:** When totalShares = 0, incoming funds are marked "accounted for" without distribution, causing ~50% permanent loss and unfair allocation for first re-added shareholders.

While the protocol achieves its core functionality, critical improvements in callback handling, randomness distribution, accounting invariants, and fund management are required for security and fairness.

Summary of Findings :

Vulnerability	Severity	Status
[C-01] Missing VRF callback failure recovery mechanism leads to permanent protocol deadlock	Critical	Fixed
[H-01] First shareholder allocation bypasses `totalUnclaimedProceeds` tracking leading to systematic contract insolvency	High	Fixed
[H-02} Incremental funding mechanism in `totalUnclaimedProceeds` leads to systematic accounting corruption and withdrawal failures	High	Fixed
[H-03] Identical modulo operations in both branches of transformRandomToPixel leads to biased lottery draws	High	Fixed
[M-01] Missing fund distribution when totalShares is zero leads to permanent fund loss and wrong allocation	Medium	Fixed
[L-01] Missing input validation in config parameter enqueuing leads to silent configuration failures	Low	Fixed
[L-02] Unnecessary cycle timing restriction on redemptions leads to unfair user experience	Low	Fixed
[L-03] Single owner as sole point of failure enables protocol compromise	Low	Fixed
[L-04] Use of Ownable instead of Ownable2Step enables accidental permanent loss of admin control	Low	Fixed
[I-01] Missing input validation in pixelToBits() leads to undefined behavior and invalid bitmap generation	Info	Fixed

7. Audit Findings

7.1 Critical Severity Findings

[C-01] Missing VRF callback failure recovery mechanism leads to permanent protocol deadlock

Description

The protocol implements an asynchronous two-transaction draw process where `draw()` initiates a VRF request and sets `isPending = true`, while `randomNumberCallback()` completes the draw and resets the pending state. However, there is no recovery mechanism if the VRF callback transaction fails, creating a permanent deadlock condition.

The vulnerability stems from the fact that these are separate transactions:

- Transaction 1: `draw()` successfully executes and permanently sets `currentDrawRequest.isPending = true`
- Transaction 2: `randomNumberCallback()` may fail due to various reasons, leaving the pending state locked

Vulnerable Scenario:
The following steps demonstrate the deadlock:

1. User calls `draw()` which successfully executes, setting `isPending = true` and requesting VRF randomness
2. VRF system calls `randomNumberCallback()` with the random number
3. The callback fails due to external transfer failure (e.g., malicious recipient contract reverts)
4. Due to transaction atomicity, the callback reverts but the original `draw()` transaction remains committed
5. The contract is now permanently locked with `isPending = true`
6. All future `draw()` calls revert with `DrawRequestPending()` error
7. No mechanism exists to reset the pending state, causing permanent protocol deadlock

Critical Failure Points in Callback:

```
// Any of these external calls can cause total revert:
(bool success, ) = payable(teamContractAddress).call{value: teamAmount}("");
if (!success) revert TransferFailed();

(bool success, ) = payable(foundationAddress).call{value: foundationAmount}("");
if (!success) revert TransferFailed();

(bool success, ) = payable(socialProjectsAddress).call{value: socialProjectsAmount}("");
if (!success) revert TransferFailed();

(bool success, ) = payable(requestCaller).call{value: drawCallerRewardAmount}("");
if (!success) revert TransferFailed();
```

Additional failure scenarios include:

- Gas limit exhaustion due to complex callback operations
- VRF system becoming unresponsive or offline
- Interface changes in external VRF system

• Impact

•

- Complete and permanent protocol shutdown with no recovery mechanism. All user funds become permanently locked in the contract as no new draws can be initiated and no existing recovery functions exist. This represents a total loss scenario for all participants.

•

Recommendation

Implement Emergency Reset Function: Add an owner-only function to reset stuck draw requests after a reasonable timeout period:

```
uint256 public constant EMERGENCY_TIMEOUT = 24 hours;

function emergencyResetDraw() external onlyOwner {
    require(currentDrawRequest.isPending, "No pending draw");
    require(block.timestamp ⩾ currentDrawTime + EMERGENCY_TIMEOUT, "Timeout not reached");

    currentDrawRequest.isPending = false;
    currentDrawRequest.requestId = 0;
    currentDrawRequest.requestCaller = address(0);

    emit EmergencyDrawReset(block.timestamp);
}
```

Golden Grid:

Confirmed

Zealynx:

Fixed

7.2 High Severity Findings

[H-01] First shareholder allocation bypasses `totalUnclaimedProceeds` tracking leading to systematic contract insolvency

Description

The `updateShares` function contains a critical flaw in the "first shareholder" allocation logic that can lead to systematic contract insolvency. When a shareholder is added as the first shareholder (when `totalShares == newShares`), the contract allocates proceeds without properly accounting for existing obligations to previous shareholders, causing the contract to promise more funds than it actually holds.

```
if (totalShares == newShares) {
    // first shareholder - only allocate unclaimed funds
    if (address(this).balance < totalUnclaimedProceeds) revert InsufficientBalance();
    proceeds[shareholderAddress] = address(this).balance - totalUnclaimedProceeds;
    // Missing: totalUnclaimedProceeds += proceeds[shareholderAddress];
}
```

The allocation uses `totalUnclaimedProceeds` to determine "unclaimed funds" but fails to update this variable after the allocation. This creates a cascading accounting failure where each subsequent "first shareholder" allocation ignores previous obligations.

Vulnerable Scenario:

The following steps help reproduce the issue:

1. Initial State: Contract receives 100 APE
2. Alice Added: Alice becomes first shareholder, gets allocated 100 APE proceeds
3. Accounting Error: `totalUnclaimedProceeds` remains 0 instead of being updated to 100 APE
4. Alice Removed: Alice's shares set to 0, but she doesn't withdraw her 100 APE proceeds
5. New Funds: Contract receives additional 50 APE (total balance: 150 APE)
6. Bob Added: Bob becomes first shareholder
7. Broken Allocation: Bob gets `150 - 0 = 150 APE` instead of `150 - 100 = 50 APE`
8. Insolvency: Contract owes 250 APE (100 to Alice + 150 to Bob) but only has 150 APE

Impact

This vulnerability creates a scenario where the contract becomes unable to fulfill all withdrawal obligations. The contract allocates more proceeds than its actual balance can support, creating a cascading effect where totalUnclaimedProceeds becomes increasingly inaccurate with each iteration.

This leads to a bank run scenario where early withdrawers receive their funds, but later withdrawers face failed transactions due to insufficient contract balance, resulting in permanent fund loss for some shareholders.

Recommendation

Fix the accounting by properly tracking first shareholder allocations:

```
if (totalShares == newShares) {
    // first shareholder - only allocate unclaimed funds
    if (address(this).balance < totalUnclaimedProceeds) revert InsufficientBalance();
    uint256 allocation = address(this).balance - totalUnclaimedProceeds;
    proceeds[shareholderAddress] = allocation;
    totalUnclaimedProceeds += allocation; // Critical: Track the allocation
}
```

This ensures the contract maintains accurate accounting of its obligations and prevents systematic insolvency.

Golden Grid:

Confirmed

Zealynx:

Fixed

[H-02] Incremental `totalUnclaimedProceeds` funding mechanism leads to systematic accounting corruption and withdrawal failures

Description

The `withdraw(uint256 amount)` function contains a flaw in its reward tracking logic that permanently corrupts future reward calculations for users who perform partial withdrawals. When a user withdraws only a portion of their proceeds, the function incorrectly resets their reward tracking index (`userRewardPerShare`) as if they had withdrawn all funds, while simultaneously retaining the remaining proceeds in their balance.

```
function withdraw(uint256 amount) external nonReentrant {
    // ... calculate totalProceeds including pending rewards
    uint256 totalProceeds = proceeds[msg.sender] + pending;

    // Store remaining proceeds after partial withdrawal
    proceeds[msg.sender] = totalProceeds - amount;

    // BUG: Reset tracking index as if user withdrew everything
    userRewardPerShare[msg.sender] = rewardPerShare;
}
```

The contract uses a Compound-style reward distribution system where `userRewardPerShare[user]` tracks the last synchronization point for calculating pending rewards. This index should only be fully reset when `proceeds[user] == 0`. By resetting it during partial withdrawals while retaining proceeds, the system breaks the fundamental invariant of the reward tracking mechanism.

Vulnerable Scenario:

The following steps help understand the issue:

1. Alice has 100 APE total proceeds available for withdrawal
2. Alice calls `withdraw(50)` to withdraw 50 APE, leaving 50 APE in her proceeds balance
3. The contract sets `proceeds[Alice] = 50 APE` (remaining amount)
4. Bug: The contract sets `userRewardPerShare[Alice] = rewardPerShare` (full reset)
5. Contract receives new deposits, increasing the global `rewardPerShare`
6. Alice's future reward calculations only consider rewards earned after step 4
7. Result: Alice's retained 50 APE never participates in future reward distributions

Impact

This vulnerability causes users who perform partial withdrawals to permanently lose the ability to earn future rewards on their retained proceeds. The retained funds become "frozen" in the reward system, unable to participate in subsequent reward distributions. The impact compounds over time as new deposits flow into the contract, with affected users missing out on proportional rewards that should apply to their retained balances. This represents a fundamental breakdown of the reward distribution system for any user who doesn't withdraw their full balance in a single transaction.

Recommendation

Fix the reward tracking logic to properly handle partial withdrawals by preventing index reset during partial withdrawals.

```
function withdraw(uint256 amount) external nonReentrant {
    // ... existing logic ...

    proceeds[msg.sender] = totalProceeds - amount;

    // Only reset index if user has no remaining proceeds
    if (proceeds[msg.sender] == 0) {
        userRewardPerShare[msg.sender] = rewardPerShare;
    }
    // If proceeds remain, keep the existing index to maintain proper tracking
}
```

In this way, it maintains the intended functionality while fixing the core accounting flaw.

Golden Grid:

Confirmed. A test was implemented and it showed that a fix included on a previous issue, fixed this vulnerability as well.

Zealynx:

Fixed. Made sure that the issue was present on commit under scope and then fixed as per dev team's test results.

[H-03] Identical modulo operations in both branches of transformRandomToPixel leads to biased lottery draws

Description

The `transformRandomToPixel` function attempts to implement rejection sampling to ensure unbiased random number distribution but fails due to executing identical modulo operations in both the "biased" and "unbiased" branches. This completely negates the bias correction mechanism, resulting in a systematically unfair lottery system.

```
function transformRandomToPixel(uint256 randomNumber) public pure returns (uint32) {
    uint256 unbiasedRange = (type(uint256).max / 98_280) * 98_280;

    if (randomNumber >= unbiasedRange) {
        return uint32(randomNumber % 98_280);  // Same operation as below
    }

    return uint32(randomNumber % 98_280);      // Same operation as above
}
```

The function calculates an `unbiasedRange` to identify numbers that would introduce bias, but then applies the same biased modulo operation (`randomNumber % 98_280`) regardless of which branch is taken. This makes the conditional check completely meaningless.

Since `2^256` is not perfectly divisible by `98_280`, the remainder `2^256 % 98_280 = 57,896` means that pixel numbers 0-57,895 have a slightly higher probability of being selected than pixel numbers 57,896-98,279.

Vulnerable Scenario:
The following steps help understand the issue:

1. VRF generates a random number in the "biased region" (>= unbiasedRange)
2. Function detects this should be handled specially to avoid bias
3. Instead of rejecting or resampling, it applies the same biased modulo operation
4. Lower pixel numbers (0-57,895) get selected with ~0.059% higher probability
5. Over time, certain lottery participants have systematically better odds

Impact

This vulnerability fundamentally compromises the lottery's integrity by creating an unfair distribution where certain pixel numbers have measurably higher winning probabilities. The bias violates the core fairness guarantee explicitly stated in the contract documentation and undermines user trust in the system. Over millions of draws, this creates a significant and measurable advantage for participants who purchase lower-numbered pixels, constituting a breach of the lottery's promise of equal odds for all participants.

Recommendation

Proper Rejection Sampling

```
function transformRandomToPixel(uint256 randomNumber) public pure returns (uint32) {
    uint256 unbiasedRange = (type(uint256).max / 98_280) * 98_280;

    if (randomNumber ⩾ unbiasedRange) {
        // Use hash-based resampling for biased region
        uint256 newRandom = uint256(keccak256(abi.encode(randomNumber)));
        return transformRandomToPixel(newRandom);
    }

    return uint32(randomNumber % 98_280);
}
```

This will ensure mathematical correctness and maintain the original rejection sampling intent, ensuring accurate uniform distribution across all 98,280 possible pixel combinations

Golden Grid:

Confirmed. We proposed a solution that differs from the proposed one to avoid infinite loops. Hence, restricting the loop to 5 retries.

Zealynx:

Fixed. Agreed with the proposed solution.

7.3 Medium Severity Findings

[M-01] Missing fund distribution when totalShares is zero leads to permanent fund loss and wrong allocation

Description

The `_sync()` function in DPLTeam contract updates `accountedBalance` regardless of whether funds can be distributed to shareholders. When funds arrive while `totalShares = 0` (no active shareholders), these funds are marked as "accounted for" but never allocated to the reward distribution system, creating both permanent fund loss and unfair distribution when shareholders are re-added.

Vulnerable Scenario:

The following steps demonstrate the dual impact of fund loss and theft:

1. Initial Setup : Owner sets up shareholders (Alice: 100 shares, Bob: 50 shares, total: 150 shares)
2. Normal Operation : 150 APE arrives and distributes correctly (Alice: 100 APE, Bob: 50 APE)
3. Shareholder Removal : Owner temporarily removes all shareholders by setting their shares to 0 (`totalShares = 0`) during team restructuring
4. Vulnerable Period : PixelLotteryAPE continues operating and sends 200 APE to DPLTeam contract while `totalShares = 0`
5. Broken Accounting : `_sync()` function executes: `accountedBalance` is updated to include the 200 APE, but `rewardPerShare` remains unchanged since `totalShares = 0`
6. First Shareholder Advantage : Owner re-adds Alice first (200 shares) - the "first shareholder" logic allocates `address(this).balance - totalUnclaimedProceeds`, giving Alice unfair access to funds
7. Second Shareholder Disadvantage : Owner adds Bob (100 shares) - Bob retains only his original 50 APE proceeds, missing his proportional share of the 200 APE
8. Permanent Loss : Some funds become permanently inaccessible due to broken accounting synchronization

Concrete Impact from Proof of Concept:
- 200 APE sent during `totalShares = 0`
- Expected fair distribution : Alice should get ~233 APE total (100 original + 133 proportional), Bob should get ~117 APE total (50 original + 67 proportional)
- Actual unfair result : Alice gets 240 APE (+7 APE surplus), Bob gets 70 APE (-47 APE deficit)
- 100 APE permanently lost and inaccessible to any shareholder
- Net effect : 50% fund loss + unfair redistribution favoring first shareholder

```
function _sync() internal {
    uint256 currentBalance = address(this).balance;
    if (currentBalance > accountedBalance) {
        uint256 delta = currentBalance - accountedBalance;
        if (totalShares > 0) {
            rewardPerShare += (delta * MULTIPLIER) / totalShares;
        }
        accountedBalance = currentBalance; // ← Always updated, even when funds not distributed
    }
}
```

Impact

1. Permanent fund loss : Approximately 50% of funds arriving during zero-shareholder periods become permanently inaccessible
2. Economic manipulation : First shareholder added back receives unfair allocation, enabling potential insider advantage
3. Broken proportional distribution : Core contract functionality compromised, violating fair share-based distribution principle

Recommendation

Only update `accountedBalance` when funds are actually distributed to prevent accounting discrepancies:

```
function _sync() internal {
    uint256 currentBalance = address(this).balance;
    if (currentBalance > accountedBalance) {
        uint256 delta = currentBalance - accountedBalance;
        if (totalShares > 0) {
            rewardPerShare += (delta * MULTIPLIER) / totalShares;
            accountedBalance = currentBalance; // Only update when distributed
        }
        // If totalShares == 0, leave accountedBalance unchanged
        // so funds remain "unaccounted" and will be distributed later
    }
}
```

Golden Grid:

Confirmed

Zealynx:

Fixed

7.4 Low Severity Findings

[L-01] Missing input validation in config parameter enqueuing leads to silent configuration failures

Description

The `enqueueConfigChange` function accepts configuration parameter changes without any validation at enqueue time. Invalid parameter values are only validated during the callback execution in `_applyPendingConfigChanges()`, where they are silently ignored rather than rejected. This creates a scenario where administrators can unknowingly submit incorrect values that either fail silently or cause unintended protocol behavior.

The following steps demonstrate the issue:

1. Admin submits a config change with an invalid value (e.g., price below minimum bounds)
2. The `enqueueConfigChange` function accepts this value without validation
3. During the next VRF callback, the invalid value is silently ignored in the setter function
4. The admin believes the change was applied but the protocol continues with old values
5. This creates a mismatch between expected and actual protocol configuration

Recommendation

1. Implement validation in `enqueueConfigChange` : Add input validation that reverts immediately for invalid parameter values, providing clear feedback to administrators.

2. Add parameter bounds documentation : Clearly document acceptable ranges for all configurable parameters.

Golden Grid:

Confirmed

Zealynx:

Fixed

[L-02] Unnecessary cycle timing restriction on redemptions leads to unfair user experience

Description

The `redeemPixels()` function incorrectly applies the `onlyDuringCycle` modifier, which prevents users from redeeming their winning pixels during the 1-hour downtime period before each draw. This restriction serves no technical security purpose since redemptions operate entirely on historical data from previous completed draws, not current cycle state.

The `onlyDuringCycle` modifier is designed to prevent new pixel purchases during the final hour before a draw to maintain cycle integrity. However, applying this same restriction to redemptions creates artificial barriers for legitimate winners.

The redemption process validates pixel ownership against `previousDrawTime` and utilizes finalized results from completed draws, ensuring it is entirely independent of current cycle timing.

Unfair user experience where legitimate winners are artificially prevented from claiming rewards, creating opportunities for economic manipulation where informed users redeem early while uninformed users face lockout periods.

Recommendation:

Remove Unnecessary Modifier: Remove the `onlyDuringCycle` modifier from the `redeemPixels()` function:

```
function redeemPixels(uint32[] calldata _pixels) external nonReentrant {
    // Remove onlyDuringCycle modifier - redemptions use historical data only
    uint32 length = uint32(_pixels.length);
    // ... rest of function remains unchanged
}
```

Golden Grid:

Confirmed

Zealynx:

Fixed. Asked for clarification on the solution implemented since it differed from the proposed one, and agreed to the reasoning for it.

[L-03] Single owner as sole point of failure enables protocol compromise

Description

The protocol implements a single owner model using OpenZeppelin's `Ownable` pattern, creating a critical single point of failure despite having multisig infrastructure available. All administrative functions, including those that can redirect protocol funds or manipulate economic parameters, are controlled by a single `onlyOwner` modifier.

While the documentation indicates the owner is intended to be a multisig wallet, the current implementation does not enforce or leverage multisig capabilities at the smart contract level. This creates an unnecessary centralization risk where:

- The compromise of the owner account (whether EOA or multisig) provides complete protocol control
- No granular access control exists for different risk levels of operations
- The multisig infrastructure is underutilized, providing no additional security layers

The risk is compounded by the fact that owner-controlled functions can:

- Redirect all protocol revenue by changing distribution addresses
- Manipulate economic parameters like pricing
- Control all aspects of protocol operation without any checks or balances

Recommendation

Leverage the existing multisig infrastructure by implementing role-based access control that utilizes multisig capabilities:

1. Replace single owner with role-based system:

```
contract PixelLotteryAPE is AccessControl {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant OPERATOR_ROLE = keccak256("OPERATOR_ROLE");
}
```

2. Assign different multisig wallets to different roles:

- ADMIN_ROLE : High-security multisig for fund-affecting functions
- OPERATOR_ROLE : Operational multisig for routine parameter changes

3. Implement role-specific function access:

```
function enqueueConfigChange(ConfigParam param, uint256 value) external {
    if (_isCriticalParameter(param)) {
        require(hasRole(ADMIN_ROLE, msg.sender), "Admin role required");
    } else {
        require(hasRole(OPERATOR_ROLE, msg.sender), "Operator role required");
    }
    // ... function logic
}
```

```
function _isCriticalParameter(ConfigParam param) private pure returns (bool) {
    return param == ConfigParam.TEAM_CONTRACT_ADDRESS ||
           param == ConfigParam.FOUNDATION_ADDRESS ||
           param == ConfigParam.SOCIAL_PROJECTS_ADDRESS;
}
```

4. Utilize multisig threshold differences:
   - Configure admin multisig with higher threshold (e.g., 4/5 signatures)
   - Configure operator multisig with lower threshold (e.g., 2/3 signatures)

This approach eliminates the single point of failure while making practical use of the available multisig infrastructure, providing defense in depth without adding significant operational overhead.

Golden Grid:

Confirmed. As agreed, we will use only `AccessControl` on lottery contract to not over-complicate things.

Zealynx:

Fixed. Discussed and agreed about the solution proposed by dev team.

[L-04] Use of Ownable instead of Ownable2Step enables accidental permanent loss of admin control

Description

Both contracts use OpenZeppelin's basic `Ownable` pattern instead of the safer `Ownable2Step` variant. This creates a risk of permanent loss of administrative control through accidental ownership transfer to an incorrect or unusable address.

The basic `Ownable` pattern allows single-transaction ownership transfer via `transferOwnership()`, which immediately and irreversibly transfers control. If the new owner address is incorrect (due to typos, wrong network, or targeting a contract that cannot call admin functions), the protocol becomes permanently unmanageable.

Given that both contracts have critical owner-controlled functions that manage fund distribution and protocol parameters, loss of ownership would result in:

- Inability to update protocol configuration
- Inability to change fund distribution addresses
- Inability to manage team shares
- Potential permanent freezing of administrative capabilities

Recommendations:

Replace `Ownable` with `Ownable2Step` in both contracts:

```
import {Ownable2Step} from "@openzeppelin/contracts/access/Ownable2Step.sol";

contract DPLTeam is Ownable2Step, ReentrancyGuard {
    constructor() Ownable(msg.sender) {}
}

contract PixelLotteryAPE is ReentrancyGuard, Ownable2Step, IVRFSystemCallback {
    constructor(...) Ownable(msg.sender) {}
}
```

The `Ownable2Step` pattern requires a two-step process:

1. Current owner calls `transferOwnership(newOwner)` (ownership becomes pending)
2. New owner must call `acceptOwnership()` to confirm the transfer

This prevents accidental permanent loss of control by ensuring the new owner address is valid and accessible before the transfer is finalized.

Golden Grid:

Confirmed. Will use Ownable2Step for DPLTeam and Role-based AccessControl for the main lottery contract.

Zealynx:

Fixed. Agreed with the discussed solution.

7.5 Informational Findings

[I-01] Missing input validation in pixelToBits() leads to undefined behavior and invalid bitmap generation

Description

The `pixelToBits()` function in PixelToBitmapAPELib lacks input validation to ensure the pixel parameter is within the valid range (0 to MAX_PIXEL_NUMBER = 98,279). While the function documentation states it accepts pixels "0 to MAX_PIXEL_NUMBER", there is no runtime check to enforce this constraint.

The function is declared `public`, making it accessible to any external caller, including future integrations, off-chain tools, and other contracts that may not implement proper input validation.

Recommendation

Add input validation to `pixelToBits()`:

```
function pixelToBits(uint32 pixel) public pure returns (uint32 bitmap) {
    // Add input validation
    if (pixel > MAX_PIXEL_NUMBER) revert InvalidInputNumber();

    // ... rest of function
}
```

Golden Grid:

Acknowledged.

Zealynx:

Advised to implement it for extra caution.