ZK ARCADE

Security Assessment

September 2025

FUZZING LABS

Contents

About FuzzingLabs

Founded in 2021 and headquartered in Paris, FuzzingLabs is a cybersecurity startup specializing in vulnerability research, fuzzing, and blockchain security. We combine cutting-edge research with hands-on expertise to secure some of the most critical components in the blockchain ecosystem.

At FuzzingLabs, we aim to uncover and mitigate vulnerabilities before they can be exploited. Over the past year, our tools and methodologies have identified hundreds of vulnerabilities in essential blockchain components, such as RPC libraries, cryptographic systems, compilers, and smart contracts. We collaborate with leading protocols and foundations to deliver open-source security tools, continuous audits, and comprehensive fuzzing services that help secure the future of blockchain technology.

If you're interested, we have a blog available at fuzzinglabs.com and an X account @FuzzingLabs. You can also contact us at contact@fuzzinglabs.com.

Executive Summary

Goals

The primary goal of this audit was to assess the security posture of the ZK Arcade smart-contract layer, with emphasis on the correctness of the mint → play → prove → verify → claim flow, leaderboard invariants, upgrade mechanism, and access control. We aimed to validate on-chain logic and state transitions, especially edge cases that could enable incorrect scoring, premature claims, or privileged misuse.

The objectives were structured into two phases:

Quick Dive

A rapid pass to surface obvious issues and gain familiarity with the repo, infrastructure and runtime configuration: run static checks, review deployment/upgrade scripts, and map the app-level threat model by reading the code (note: there were no unit tests).

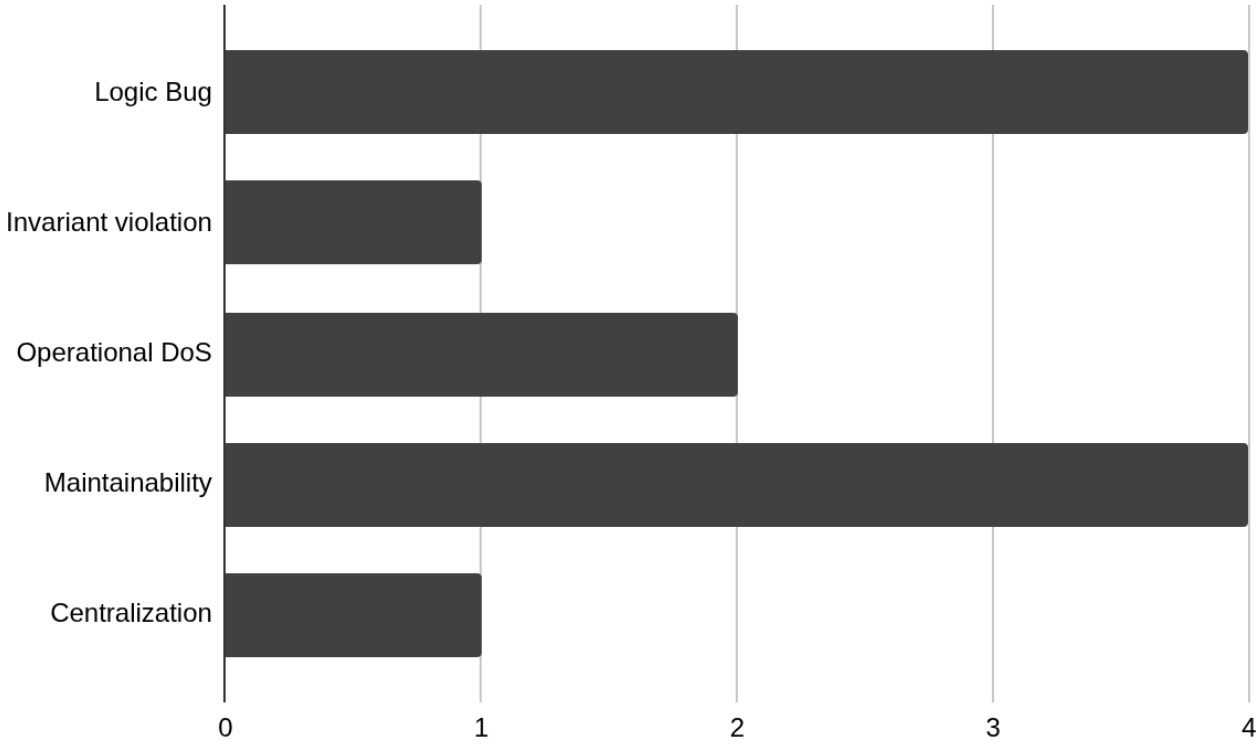
Deep Dive

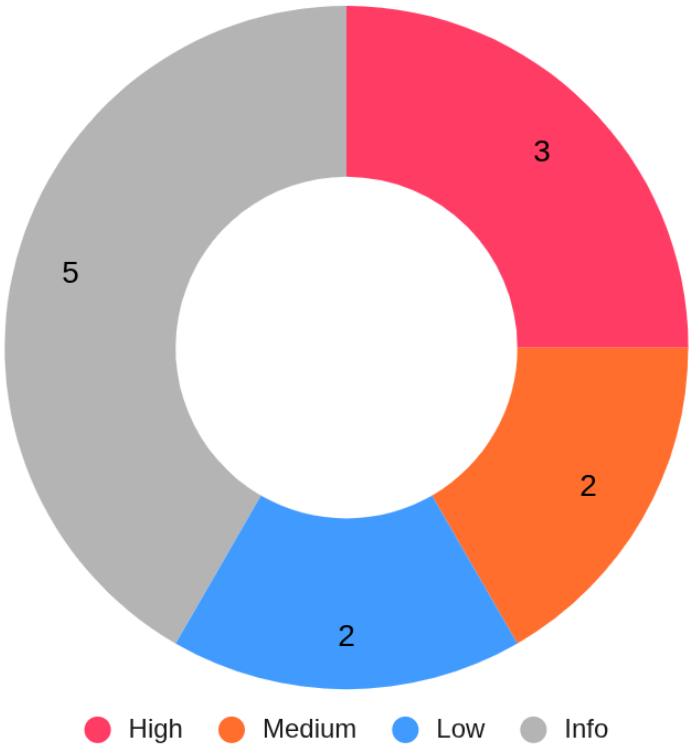
A focused, line-by-line review of Leaderboard.sol, ZkArcadeNft.sol, and ZkArcadePublicNft.sol (along with their associated Solidity scripts) to detect subtle logic and validation gaps. Primary targets were access-control correctness (who can set VKs, merkle roots, award points, etc.), claim flows (claimBeastPoints / claimParityPoints), Merkle handling, non-transferable token semantics, and leaderboard ordering invariants. This phase included targeted PoCs and invariant tests for leaderboard properties to confirm exploitability of high-impact issues.

Limitations

The audit of ZK Arcade was conducted with full white card access, providing our team unrestricted visibility into the system's architecture, codebase, and operational environment.

Summary of Findings





ID	Title	Target	Rating
1	Incorrect early return allows corruption of Top-10 Leaderboard ordering	Leaderboard	High
2	Missing check in claimBeastPoints allowing claims before a game started	Leaderboard	High
3	Missing check in claimParityPoints allowing claims before a game started	Leaderboard	High
4	Reverse-loop underflow in getCurrentBeastGame	Leaderboard	Medium
5	Reverse-loop underflow in getCurrentParityGame	Leaderboard	Medium
6	Potential DoS by transfer-before-mint in ZkArcadePublicNft::mint	ZkArcadePublicNft	Low
7	Unvalidated Aligned Service addresses	Leaderboard	Low
8	shiftAmount underflow in Parity verification	Leaderboard	Info
9	Unused IERC721 Import in ZkArcadeNft	ZkArcadeNft	Info
10	Centralization Risk in setBeastVkCommitment / setParityVkCommitment	Leaderboard	Info
11	Transfer pause check in transferFrom	ZkArcadeNft / ZkArcadePublicNft	Info
12	Multiple packed hashing used across the code	Leaderboard	Info

Project Summary

About Aligned Layer & ZK Arcade

Aligned Layer is a decentralized network designed to provide fast, efficient, and low-cost verification of zero-knowledge (ZK) and validity proofs on the Ethereum blockchain. It seeks to overcome the limitations of current blockchain verification systems, which are often slow and expensive due to the need for nodes to re-execute each transaction. By offloading the computation off-chain, Aligned Layer enables faster proof verification and significantly reduces costs.

ZK Arcade is an on-chain gaming demo built on top of Aligned where players mint a participation NFT, play off-chain puzzles/games, generate ZK proofs of their score, and submit those proofs for verification to claim leaderboard points. The public demo site and repo present the gameplay flow (mint → play → prove → verify → claim) and explicitly note that proofs are verified via Aligned to keep verification fast and affordable.

Scope

This audit was performed on commit:

6af1edb72fd32bc86dd6d7c66853eca029f6bcd5

The review focused primarily on the smart-contract layer of the ZK Arcade project. While we inspected repository-level deployment and helper scripts to understand expected on-chain configuration and upgrade flows, the bulk of the effort targeted the three Solidity contracts that implement token, eligibility and leaderboard functionality.

The scope of analysis included:

Smart contracts (core)

- `src/Leaderboard.sol`
  - Access control and owner/deployer privileges.
  - Claim / point-awarding logic and score accounting.
  - Interaction with proof-verification / claim entry points, input validation and replay/duplication protections.
  - Upgradeability considerations (UUPS patterns if present).
  - Event emission, state transitions, and edge cases around repeated claims or malformed inputs.

- `src/ZkArcadeNft.sol`
  - Minting/claim flow and `hasClaimed` bookkeeping.
  - Merkle root handling and merkle-related functions.
  - Transferability flags (`transfersEnabled`) and intended non-transferable / soulbound behavior.
  - Owner-only functions, timelocks or guards, and potential for privileged misuse.

- `src/ZkArcadePublicNft.sol`
  - Differences vs `ZkArcadeNft` (public vs gated behavior).
  - Access-control, minting rules, and interactions with the leaderboard/claim flow.
  - Any divergences in transfer / approval semantics.

Deployment tooling

- `scripts/` and `script/` (deploy, upgrade, helper scripts) - reviewed to verify:
  - Correct constructor/initializer parameters are used.
  - Merkle root / campaign management scripts behave as intended and do not introduce accidental privileged state changes.
  - That the on-chain deployment flow described in scripts matches the contracts' expectations (ownership, initial flags).

Out of scope

- Off-chain components and infrastructure (Aligned Layer, Aligned's verification network, relayers or aggregator services).
- Client/front-end code and UI behavior beyond confirming the intended user flow.
- External ZK circuits, prover implementations, and proof-generation toolchains.
- Third-party services, validator/restaking security model, and any external protocol assumptions (those belong to Aligned's threat model).

Audit Timeline

The security assessment was conducted by **Dimitri Carlier** over a total of **7 man-days**. The work was structured as follows:

- **Context & Threat-Model Review (1 day)**
  - An initial pass through the whole repository and deployment scripts was performed to understand the full application flow. This review was specifically focused on comprehending the app logic and attacker capabilities relevant to the contracts.
- **Static Analysis & Repository Review (1 day)**
  - Automated static checks and linters were run to identify obvious weaknesses and code anti-patterns. Because **no unit tests** were present in the repo, we performed manual smoke checks of scripts and contract entrypoints to validate expected behaviors and collect a shortlist of quick, high-priority items for manual follow-up.
- **Manual Code Review, Detailed Reports and Proof-of-Concepts (4 days)**
  - A detailed, line-by-line review of the Solidity contracts (Leaderboard.sol, ZkArcadeNft.sol, ZkArcadePublicNft.sol) and relevant deployment/upgrade scripts was carried out.
  - Focus areas included access control, claim/score logic, Merkle handling, UUPS upgrade mechanics, and soulbound token semantics. Targeted proof-of-concepts (PoCs) and test snippets were developed where necessary to confirm exploitability and impact for findings.
- **Invariants Testing for the Leaderboard (1 day)**
  - Dedicated invariant tests were written and executed for the leaderboard to validate core properties.

This structured approach provided both breadth (repo and script review) and depth (manual contract auditing and PoCs) on the highest-impact components of the project.

Access Control and Invariants

Contracts

Leaderboard.sol

Function	Who can call	Auth	State / Preconditions
initialize(...)	Deployer (proxy)	initializer	One-time init
claimBeastPoints(...)	Any EOA	none	msg.sender must match encoded user; if useWhitelist==true user must hold NFT; Aligned verification must succeed; game not ended; config matches; level strictly increases
claimParityPoints(...)	Any EOA	none	msg.sender must match encoded user; if useWhitelist==true user must hold NFT; Aligned verification must succeed; parity game not ended; shifted config matches; level strictly increases
getUserScore(...)	Anyone	view	—
getUserBeastLevelCompleted(...)	Anyone	view	—
getCurrentBeastGame()	Anyone	view	Reverts if none active
getCurrentParityGame()	Anyone	view	Reverts if none active
getTop10Score()	Anyone	view	—
setBeastGames(...)	Owner	onlyOwner	—
addBeastGames(...)	Owner	onlyOwner	—
setParityGames(...)	Owner	onlyOwner	—
addParityGames(...)	Owner	onlyOwner	—

enableWhitelist()	Owner	onlyOwner	—
disableWhitelist()	Owner	onlyOwner	—
setZkArcadeNftAddress(...)	Owner	onlyOwner	—
setZkArcadePublicNftAddress(...)	Owner	onlyOwner	—
setBeastVkCommitment(...)	Owner	onlyOwner	—
setParityVkCommitment(...)	Owner	onlyOwner	—

Invariants:

- Top10 ordering:
  - `usersScore(top[i]) >= usersScore(top[i+1]) for all i in 0..8`
- No zero before non-zero (zero-tail compactness):
  - `if top[j] == address(0) then for all k > j: top[k] == address(0)`
- No duplicates:
  - `for all i != j: if top[i] != address(0) and top[j] != address(0) then top[i] != top[j]`
- Non-zero entries have positive score:
  - `for all i: if top[i] != address(0) then usersScore(top[i]) > 0`

ZkArcadeNft.sol

Function	Who can call	Auth	State / Preconditions
initialize(...)	Deployer (proxy)	initializer	One-time init; sets claimsEnabled=true, transfersEnabled=false
claimNFT(...)	Any EOA	none	claimsEnabled==true; !hasClaimed[msg.sender]; rootIndex in range; Merkle proof valid
transferFrom(...)	Approved / operator	override	transfersEnabled==true else revert TransfersPaused()
isWhitelisted(...)	Anyone	view	—
addMerkleRoot(...)	Owner	onlyOwner	—
setMerkleRoot(...)	Owner	onlyOwner	idx in range
enableTransfers()	Owner	onlyOwner	—
disableTransfers()	Owner	onlyOwner	—
enableClaims()	Owner	onlyOwner	—
disableClaims()	Owner	onlyOwner	—
endSeason()	Owner	onlyOwner	—
setBaseURI(...)	Owner	onlyOwner	—

ZkArcadePublicNft.sol

Function	Who can call	Auth	State / Preconditions
initialize(...)	Deployer (proxy)	initializer	One-time init; sets mintingEnabled=false, transfersEnabled=false
mint()	Any EOA	none	mintingEnabled==true; caller must have balanceOf(msg.sender) == 0; _nextTokenId < maxSupply
transferFrom(...)	Approved / operator	override	transfersEnabled==true else revert TransfersPaused()
totalSupply()	Anyone	view	—
enableMinting()	Owner	onlyOwner	—
disableMinting()	Owner	onlyOwner	—
enableTransfers()	Owner	onlyOwner	—
disableTransfers()	Owner	onlyOwner	—
setBaseURI(...)	Owner	onlyOwner	—

Findings

I. Incorrect early return allows corruption of Top-10 Leaderboard ordering

Rating	**High**
ID	**FL-ZA-1**
Category	**Invariant violation / Logic bug**
Target	**Leaderboard**
Status	**Fixed**

Description

The verifyAndReplaceInTop10 function contains an incorrect early-return that prevents repositioning a user who is already at index 9 when their score increases. When a user in the 10th slot updates their score upward, the function compares the new user score to the score at top10Score[9]. If top10Score[9] == user that comparison is against the same value (equality) and the function returns early, leaving the leaderboard unsorted **breaking the invariant**. This can cause problems in downstream logic that assumes the top10 is sorted (e.g., payouts, rewards, promotions), leading to incorrect behavior or potential exploits.

Guilty code snippet in verifyAndReplaceInTop10:

```javascript
uint256 userScore = usersScore[user];
// early return to not run the whole alg if the user does not
have enough points to be in the top 10
if (userScore <= usersScore[top10Score[9]]) {
    return;
}
```

This is incorrect when the user being updated is currently sitting at index 9. After a claim that increases usersScore[user], the comparison becomes userScore <= usersScore[user] (true by equality), and the function returns without repositioning the entry.

Recommendation

A minimal safe fix is to only early-return when the user is *not* the current last element. Also guard address(0):

JavaScript

```
function verifyAndReplaceInTop10(address user) internal {
    uint256 userScore = usersScore[user];

    // protect against empty slot
    uint256 lastScore = top10Score[9] == address(0) ? 0 :
usersScore[top10Score[9]];

    // early return only when user is not already in last slot
and doesn't beat last slot score
    if (top10Score[9] != user && userScore <= lastScore) {
        return;
    }

    ...
}
```

II. Missing check in claimBeastPoints allowing claims before a game started

Rating	**High**
ID	**FL-ZA-2**
Category	**Logic bug**
Target	**Leaderboard**
Status	**Fixed**

Description

The claimBeastPoints function only check that the game has not ended (block.timestamp >= game.endsAtTime). But It do **not** ensure block.timestamp >= startsAtTime. The contract therefore accepts valid proofs for game progress even outside the scheduled active window.

From claimBeastPoints:

```javascript
BeastGame memory game = beastGames[gameIndex];
// no check for startsAtTime
if (block.timestamp >= game.endsAtTime) {
    revert GameEnded();
}
if (game.gameConfig != gameConfig) {
    revert InvalidGame(game.gameConfig, gameConfig);
}
```

Recommendations

Enforce that claims occur only while the game is active: require block.timestamp >= startsAtTime **and** block.timestamp < endsAtTime. Add a custom error (e.g. GameNotStarted()) for clarity.

```javascript
error GameNotStarted();

// In claimBeastPoints, after loading game:
if (block.timestamp < game.startsAtTime) revert
GameNotStarted();
if (block.timestamp >= game.endsAtTime) revert GameEnded();
```

III. Missing check in claimParityPoints allowing claims before a game started

Rating	**High**
ID	**FL-ZA-3**
Category	**Logic bug**
Target	**Leaderboard**
Status	**Fixed**

Description

The claimParityPoints function only check that the game has not ended (block.timestamp >= game.endsAtTime). But It do **not** ensure block.timestamp >= startsAtTime. The contract therefore accepts valid proofs for game progress even outside the scheduled active window.

From claimParityPoints:

```javascript
ParityGame memory currentGame = parityGames[gameIndex];
// no check for startsAtTime
if (block.timestamp >= game.endsAtTime) {
    revert GameEnded();
}
...
```

Recommendations

Enforce that claims occur only while the game is active: require block.timestamp >= startsAtTime **and** block.timestamp < endsAtTime. Add a custom error (e.g. GameNotStarted()) for clarity.

```javascript
error GameNotStarted();

// In claimParityPoints, after loading game:
if (block.timestamp < game.startsAtTime) revert
GameNotStarted();
if (block.timestamp >= game.endsAtTime) revert GameEnded();
```

IV. Reverse-loop underflow in getCurrentBeastGame

Rating	**Medium**
ID	**FL-ZA-4**
Category	**Logic Bug**
Target	**Leaderboard**
Status	**Fixed**

Description

In getCurrentBeastGame, there is tautology/contradiction (i >= 0) because unsigned integers i >= 0 is always true and the loop becomes unsafe. When the arrays are empty or when none of the games match the timestamp, the loop underflows and the function panics rather than gracefully reverting with NoActiveBeastGame(). If callers rely on these view function in on-chain logic or other contracts, a panic can cause unexpected reverts and DoS effects.

Code snippet:

```javascript
function getCurrentBeastGame() public view returns (BeastGame
memory, uint256 idx) {
    for (uint256 i = beastGames.length - 1; i >= 0; i--) {
        if (block.timestamp >= beastGames[i].startsAtTime &&
block.timestamp < beastGames[i].endsAtTime) {
            return (beastGames[i], i);
        }
    }
    // never reached due to underflow
    revert NoActiveBeastGame();
}
```

Recommendations

Replace reverse loops with the safe reverse-loop pattern that avoids underflow by iterating from `len` down to `1` and computing `j = i - 1`. This guarantees no underflow and ensures `revert NoActiveBeastGame()` is reached when applicable.

JavaScript

```
function getCurrentBeastGame() public view returns (BeastGame
memory, uint256 idx) {
    uint256 len = beastGames.length;
    for (uint256 i = len; i > 0; i--) {
        uint256 j = i - 1;
        BeastGame memory g = beastGames[j];
        if (block.timestamp >= g.startsAtTime &&
block.timestamp < g.endsAtTime) {
            return (g, j);
        }
    }
    revert NoActiveBeastGame();
}
```

V. Reverse-loop underflow in getCurrentParityGame

Rating	**Medium**
ID	**FL-ZA-5**
Category	**Logic bug**
Target	**Leaderboard**
Status	**Fixed**

Description

In getCurrentParityGame, there is tautology/contradiction (i >= 0) because unsigned integers i >= 0 is always true and the loop becomes unsafe. When the arrays are empty or when none of the games match the timestamp, the loop underflows and the function panics rather than gracefully reverting with NoActiveParityGame(). If callers rely on these view function in on-chain logic or other contracts, a panic can cause unexpected reverts and DoS effects.

Code snippet:

```javascript
function getCurrentParityGame() public view returns (ParityGame
memory, uint256 idx) {
    for (uint256 i = parityGames.length - 1; i >= 0; i--) {
        if (block.timestamp >= parityGames[i].startsAtTime &&
block.timestamp < parityGames[i].endsAtTime) {
            return (parityGames[i], i);
        }
    }
    // never reached due to underflow
    revert NoActiveParityGame();
}
```

Recommendations

Replace reverse loops with the safe reverse-loop pattern that avoids underflow by iterating from `len` down to `1` and computing `j = i - 1`. This guarantees no underflow and ensures `revert NoActiveParityGame()` is reached when applicable.

JavaScript

```
function getCurrentParityGame() public view returns (ParityGame
memory, uint256 idx) {
    uint256 len = parityGames.length;
    for (uint256 i = len; i > 0; i--) {
        uint256 j = i - 1;
        ParityGame memory g = parityGames[j];
        if (block.timestamp >= g.startsAtTime &&
block.timestamp < g.endsAtTime) {
            return (g, j);
        }
    }
    revert NoActiveParityGame();
}
```

VI. Potential DoS by transfer-before-mint in ZkArcadePublicNft::mint

Rating	**Low**
ID	**FL-ZA-6**
Category	**Operational DoS**
Target	**ZkArcadePublicNft**
Status	**Risk Accepted**

Description

The public mint enforces a “one token per wallet” check by inspecting balanceOf(msg.sender) before minting. This enables two practical griefing vectors:

- **Front-run / transfer-right-before-mint**: an attacker transfers a token into a victim wallet immediately before the victim’s mint tx is validated, causing the victim’s mint to revert.
- **Pre-transfer (inter-season)**: if transfers are allowed during the off-season, an attacker can pre-send a token to a future participant; when minting reopens, that participant is blocked from minting because balanceOf > 0.

Both are DoS / grief on individual users (not theft), waste gas and can prevent users from participating in the campaign.

Recommendations

If you want to allow transfers during minting and avoid DoS, replace balanceOf check with a hasMinted check (which is consistent with the behavior of ZkArcadeNft). That guarantees a wallet can only mint once, regardless of transfers.

```javascript
mapping(address => bool) public hasMinted;

function mint() public returns (uint256) {
    if (!mintingEnabled) revert MintingPaused();
    if (hasMinted[msg.sender]) revert AlreadyMinted();
    if (_nextTokenId >= maxSupply) revert MaxSupplyExceeded();

    hasMinted[msg.sender] = true;
    uint256 tokenId = _nextTokenId++;
    _mint(msg.sender, tokenId);
    emit NFTMinted(msg.sender, tokenId);
    return tokenId;
}
```

Note: *a wallet could end up with 2 tokens (one minted + one transferred) if they receive a transfer.*

VII. Unvalidated Aligned Service addresses

Rating	Low
ID	**FL-ZA-7**
Category	**Operational DoS**
Target	**Leaderboard**
Status	Fixed

Description

Leaderboard::initialize assigns _alignedServiceManager and _alignedBatcherPaymentService directly to storage (lines ~72–73) with no zero-address or "is contract" checks. These addresses are later relied on for a low-level staticcall and decoded return data; a bad/missing address can cause claim calls to revert and permanently break the on-chain claiming flow because there is no way to change them after init.

Recommendations

Validate inputs at initialization: require non-zero + that alignedServiceManager && alignedBatcherPaymentService are contracts, and then store them. Minimal patch:

JavaScript

```
// in initialize
require(_alignedServiceManager != address(0) &&
_alignedServiceManager.code.length > 0, "invalid
alignedServiceManager");
require(_alignedBatcherPaymentService != address(0) &&
_alignedBatcherPaymentService.code.length > 0, "invalid
alignedBatcherPaymentService");
alignedServiceManager = _alignedServiceManager;
alignedBatcherPaymentService = _alignedBatcherPaymentService;
```

VIII. shiftAmount underflow in Parity verification

Rating	Informational
ID	FL-ZA-8
Category	Maintainability
Target	Leaderboard
Status	Fixed

Description

claimParityPoints computes shiftAmount = 256 - (80 * levelCompleted) to compare the prefix of gameConfig, thus **reverting** if levelCompleted >= 4.

Note: Given the circuit **MAX_LEVELS = 3** (component main = ValidateParityGame(3, 55)) and the Rust packer encodes exactly 3 levels x 10 bytes = 30 bytes (with 2 leading zero bytes), any proof that verifies against the current vk can only produce levelCompleted ∈ {0,1,2,3}.

Circuit enforces max 3 levels:

```C/C++
template ValidateParityGame(MAX_LEVELS, MAX_ROUNDS) {
    // ...
    signal output max_level_completed;
    // ...
    for (var i = 0; i < MAX_LEVELS; i++) {
        // compute per-level completion
        level_completed_sum = level_completed_sum + 1 -
level_completed[i].out;
        // pack 10 bytes per level into acc ...
    }
    max_level_completed <== level_completed_sum;
    gameConfigPacked <-- acc;
}
```

```
// Current instantiation fixes MAX_LEVELS = 3
component main {public [userAddress]} = ValidateParityGame(3,
55);
```

Rust packer encodes exactly 3 levels (10 bytes/level) with 2-byte padding:

Rust

```rust
// 30 bytes used (3 * 10), padded with two leading zeroes in a
[u8;32]
let offset_bytes = 2;
for (i, level) in levels.iter().enumerate() {
    let i = i * 10 + offset_bytes;
    bytes[i] = initial_pos;              // 1 byte
    for (j, tile) in level.board.iter().enumerate() {
        bytes[i + j + 1] = *tile;        // +9 bytes
    }
}
```

Contract logic that could underflow if bits > 256:

JavaScript

```javascript
// Each level takes 10 bytes -> 80 bits
uint256 shiftAmount = 256 - (80 * (levelCompleted)); // can
overflow if MAX_LEVEL is changed to >=4 in the future
uint256 currentGameConfigUntil = currentGame.gameConfig >>
shiftAmount;
uint256 gameConfigUntil = gameConfig >> shiftAmount;

if (currentGameConfigUntil != gameConfigUntil) {
    revert InvalidGame(currentGame.gameConfig, gameConfig);
}
```

Given the circuit's max_level_completed is committed as a **public input** and included in the batch verification against your vkCommitment, an attacker cannot arbitrarily set levelCompleted > 3 without breaking the proof.

Result: the subtraction underflow is **unreachable** in normal operation. The only realistic impact appears if the verifier or commitments are misconfigured (e.g., a vk compiled with more than 3 levels), in which case claims could systematically revert. **BUT** if a future deployment updates the circuit (e.g., MAX_LEVELS=4), then claims with levelCompleted > 3 would pass verification but revert on-chain (panic in 256 - 80*levelCompleted), creating an availability failure until fixed.

Recommendations

This is a **defense-in-depth** hardening to prevent opaque panics if configurations change in the future.

1. Document the invariant in-code: MAX_PARITY_LEVELS = 3 (30 bytes packed at positions [2..31]).
2. Add an explicit on-chain bound so that, if a future vk allows >3 levels, the contract fails with a clear error instead of a generic underflow panic.

Patch (explicit bound + readable error)

```javascript
error LevelTooLarge();

uint256 constant MAX_PARITY_LEVELS = 3;   // Must match circuit
(ValidateParityGame(3, ...))
uint256 constant BITS_PER_LEVEL   = 80;   // 10 bytes per level

function claimParityPoints(/* ... */) public {
    // ... after proofIncluded == true and time window checks

    if (levelCompleted > MAX_PARITY_LEVELS) revert
LevelTooLarge();

    uint256 bits = BITS_PER_LEVEL * levelCompleted;     // ≤
240 guaranteed
    uint256 shiftAmount = 256 - bits;
```

```
    uint256 currentGameConfigUntil = currentGame.gameConfig >>
shiftAmount;
    uint256 gameConfigUntil = gameConfig >> shiftAmount;

    if (currentGameConfigUntil != gameConfigUntil) {
        revert InvalidGame(currentGame.gameConfig, gameConfig);
    }

    // ... rest unchanged
}
```

IX. Unused IERC721 Import in ZkArcadeNft

Rating	Informational
ID	FL-ZA-9
Category	Maintainability
Target	ZkArcadeNft
Status	Fixed

Description

The ZkArcadeNft contract imports IERC721 from OpenZeppelin but never uses it anywhere in the implementation.

Recommendation

Remove the unused IERC721 import from ZkArcadeNft.sol to keep the codebase clean and avoid confusion.

X. Centralization Risk in setBeastVkCommitment / setParityVkCommitment

Rating	Informational
ID	**FL-ZA-10**
Category	**Centralization**
Target	**Leaderboard**
Status	**Acknowledged**

Description

In the Leaderboard contract, the owner can update the verifying key commitments at any time using setBeastVkCommitment and setParityVkCommitment. Because these values are critical for proof verification, this creates a strong centralization risk: a malicious or compromised owner could deliberately alter the commitments to manipulate which proofs are accepted, while even an accidental misconfiguration would immediately break all claim functions until the commitments are corrected. This means both fairness and availability of the system fully depend on a single trusted operator.

Recommendation

Use multisig or governance for these setters, add a timelock to allow monitoring before changes take effect, and ensure operational procedures (e.g., automated tests or dry-runs) validate new commitments before they are applied on-chain.

XI. Transfer pause check in transferFrom

Rating	Informational
ID	FL-ZA-11
Category	Maintainability
Target	ZkArcadeNft / ZkArcadePublicNft
Status	PR in review

Description

The NFT contracts currently block transfers by checking transfersEnabled only inside transferFrom. In OpenZeppelin v5 this is functionally sufficient because safeTransferFrom calls transferFrom, so both safe and unsafe transfers are blocked. However, putting the gate only in transferFrom is a maintenance risk: future refactors, custom transfer paths, or direct calls to internal _update/_transfer could unintentionally bypass the pause. Centralizing the check in a single internal update point (e.g. _update) while explicitly allowing mint/burn by checking from == address(0) or to == address(0) is a cleaner, more robust pattern.

The null/low (no immediate exploit in OZ v5) risk comes from future code changes (new helpers, direct _update usage, backports to a different OZ version) that may call internal transfer functions directly. Centralizing the check reduces the chance of accidental bypass and consolidates logic for easier auditing and less duplicate code.

JavaScript

```
// current pattern in ZkArcadeNft / ZkArcadePublicNft
function transferFrom(address from, address to, uint256
tokenId) public override {
    if (!transfersEnabled) revert TransfersPaused();
    super.transferFrom(from, to, tokenId);
}
```

Recommendation

Move the transfers-enabled guard to the internal update point and allow mint/burn by checking `from`/`to` for zero addresses:

```javascript
function _update(address to, uint256 tokenId, address auth)
internal override returns (address from) {
    from = super._ownerOf(tokenId);
    // only block actual transfers (not mint or burn)
    if (!transfersEnabled && from != address(0) && to !=
address(0)) revert TransfersPaused();
    return super._update(to, tokenId, auth);
}
```

XII. Multiple packed hashing used across the code

Rating	Informational
ID	**FL-ZA-12**
Category	**Maintainability / Gas optimization**
Target	**Leaderboard**
Status	Fixed

Description

In Leaderboard (helper functions getBeastKey / getParityKey) the key material is built using keccak256(abi.encodePacked(...)), first hashing the numeric game value with encodePacked and then packing that bytes32 together with an address. This use of abi.encodePacked across multiple fields is the root cause: while it works today for fixed-size types, it creates an avoidable packing/concatenation pattern that is fragile and can lead to ambiguous byte streams or collisions if types change or are refactored. The immediate risk is null but the pattern increases the chance of subtle bugs or future vulnerabilities; using a single canonical keccak256(abi.encode(user, game)) removes the ambiguity, saves gas and is recommended.

Recommendation

Replace the packed + double-hash pattern with a single canonical hash, e.g. return keccak256(abi.encode(user, game));, to eliminate packing ambiguity and remove the unnecessary intermediate hash.

Conclusion

The audit of the ZK Arcade smart-contracts revealed no critical vulnerabilities. It identified **three** high-severity, **two** medium-severity, **two** low-severity issues and **several** informational findings. Most of these are logical validation or boundary-edge issues that are straightforward to remediate and do not require architectural redesign. The three high-severity findings; an incorrect early return that can corrupt Top-10 leaderboard ordering and missing validation in claimBeastPoints and claimParityPoints that permit claims before a game starts; are the highest priority and should be fixed prior to mainnet deployment.

Overall, the codebase is clean and reasonably well structured. With the recommended improvements — primarily input/state validation, safer loop/index handling, explicit address validation for external services, and adding unit/invariant tests for the leaderboard and claim flows — the ZK Arcade smart-contract layer will be significantly strengthened, improving both security and reliability without changing the project's core design.

Disclaimer

This report is provided under the terms of the agreement between FuzzingLabs and the client. It is intended solely for the client's use and may not be shared or referenced without FuzzingLabs' prior written consent.
The findings in this report are based on a point-in-time assessment and do not guarantee the absence of vulnerabilities or security flaws. FuzzingLabs cannot ensure future security as systems and threats evolve. We recommend continuous monitoring, independent assessments, and a bug bounty program.
This report is not financial, legal, or investment advice, and should not be used for decision-making regarding project involvement or investment. FuzzingLabs aims to help reduce risks, but we do not provide any guarantees regarding the complete security of the technology assessed.