By
Quit

Contract Review
Issue date
11/4/2025

Overview

The following is a review of MegaweaponShop, a payment ingestion contract powering the Megaweapon game on Abstract.

Contracts in scope for this review include:

- `contracts/MegaweaponShop.sol`

This review is based on SHA `8ab0e16fd277f8a77aa2da2ad05428f24c9760ae`, and aims to identify security vulnerabilities, opportunities for gas optimization, and general best-practice recommendations with regard to the contracts in scope. The review should not be considered an endorsement of the project, nor is it a guarantee of security.

Findings

G-01: Product struct is not optimized

Severity: Gas Optimization

`Product` uses three storage slots but could be reduced to one. The struct contains two uint256 values for `priceInWei` and `credits` plus a bool `exists` flag. `exists` is redundant because product existence can be inferred by checking whether `priceInWei` is greater than zero, which is already enforced as a requirement in `addProduct`. Recommend restructuring `Product` as uint128 `priceInWei` and uint128 `credits`, removing `exists` entirely, and checking `priceInWei > 0` to determine product existence.

G-02: `ReentrancyGuard` is unnecessary

Severity: Gas Optimization

The contract uses `ReentrancyGuard` on both `purchase` and `withdraw`, but neither function is vulnerable to reentrancy attacks. `purchase` follows a checks-only pattern with no state changes or external calls, making reentrancy impossible. The withdraw function is owner gated and transfers the entire contract balance in a single operation with no state changes afterward, eliminating any reentrancy risk even if the owner were a malicious contract. Recommend removing the `ReentrancyGuard` inheritance and `nonReentrant` modifiers from both functions.

G-03: Use Solady for free optimizations

Severity: Gas Optimization

`MegaweaponShop` inherits from OpenZeppelin's `Ownable` and `ReentrancyGuard`. Consider using Solady, which has adaptations of both that are more gas efficient.

L-01: Purchase function provides no onchain proof of credit ownership

Severity: Low

The purchase function accepts ETH payments but only emits an event without updating any state to track user balances or purchase history. Recommend adding an on-chain mapping such as `mapping(address => mapping(uint256 => uint256)) public userPurchases` to track the number of times each user has purchased each product type.

Summary

`MegaweaponShop` implements a straightforward payment gateway for purchasing in-game credits using ETH on Abstract. Users send ETH to purchase credits in batches labeled as "products" as defined by the owner, with the contract emitting events consumed by an off-chain backend service to credit user accounts in the game system.

The security posture is sound with explicit validation of product existence and payment amounts, and proper access control limiting administrative functions to the owner.

The system operates with a centralized trust model where the owner has full administrative control over product offerings and fund withdrawal. Users must trust the backend service to correctly monitor events and credit their accounts, as the contract maintains no onchain record of purchases beyond event logs.

Backend Integration Note:

The `deposit-api` service contains an issue where credit values are converted to a Javascript `Number` rather than `BigInt` or `String`, creating potential precision loss for values exceeding `MAX_SAFE_INTEGER`. Recommend updating the backend to use standard Solidity-to-JavaScript patterns that preserve precision for large numbers: represent the value using a BigInt or String within JS, and a String in the db.

Additionally, the contract is designed to support future product types including season passes (where credits would be set to 0 and `productType` identifies the access tier), but the current backend implementation only extracts and processes the credits field. Before offering non-credit products like season passes, the backend will need to be adjusted to parse `productType` from events and route to appropriate handlers for time-based access grants versus in-game currency credits.

Note on access control:

The contract uses OpenZeppelin's `Ownable` to manage access control, applying `onlyOwner` to the `addProduct`, `removeProduct`, and `withdraw` functions. I recommend switching to OwnableRoles (both OZ and Solady have versions of this) and granting a backend signer with the ability to add/remove products, but leaving fund withdrawal as owner only.

The contract is production-ready for its intended use case as a payment gateway for a trusted gaming system. The contract's simplicity and focused scope limits attack surface, with the primary considerations being the trust assumptions around event monitoring and centralized owner control.