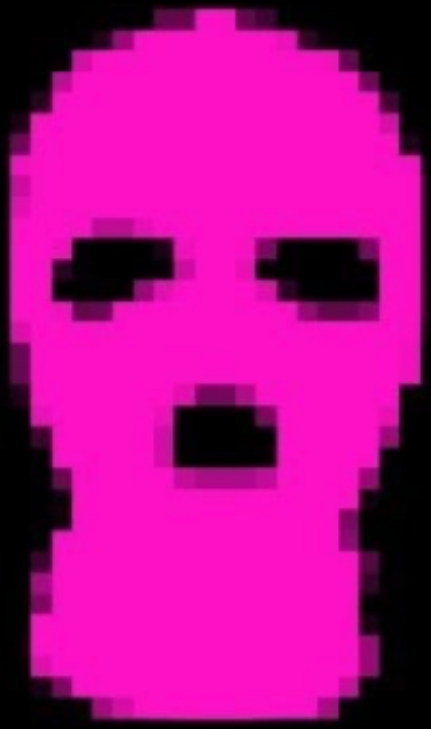
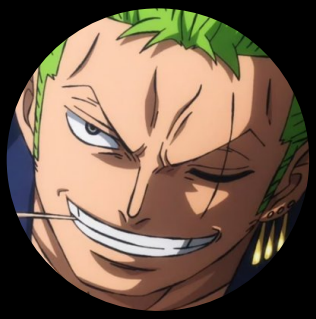
ADDICTEDDOTFUN S2
Security Review



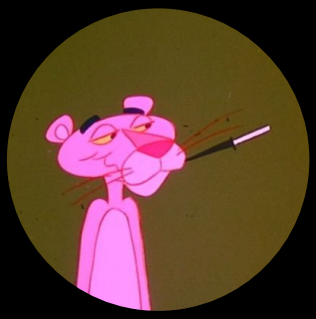
OCTOBER, 2025

www.chaindefenders.xyz
https://x.com/DefendersAudits

Lead Auditors



PeterSR



0x539.eth

Table of Contents

Protocol Summary

A blockchain-based Gacha game for Season 2 of “Addicted” using WEED tokens on Solana.

Disclaimer

The Chain Defenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the implementation of the contracts.

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

Commit Hash

fdaf0b613ec502b7a1e938f35dc1c1be64736a0d

Scope

Id	Files in scope
1	admin.rs
2	gacha.rs
3	view.rs
4	vrf.rs
5	constants.rs
6	error.rs
7	events.rs
8	lib.rs
9	state.rs

Roles

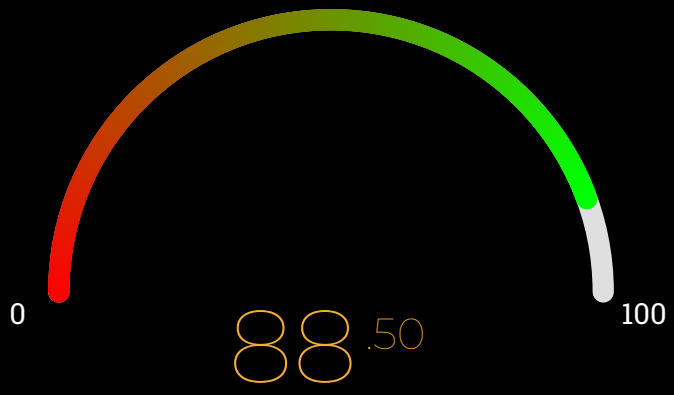
Id	Roles
1	Admin
2	User

Executive Summary

Issues Found

Severity	Count	Description
High	2	Critical vulnerabilities
Medium	2	Significant risks
Low	6	Minor issues with low impact
Informational	5	Best practices or suggestions
Gas	0	Optimization opportunities

Security Grade



Based on the audit findings and code quality, the overall protocol security grade is 88.50 / 100.00. This reflects a high security posture with small room for improvement.

Findings

High

High 01 Missing Ownership Validation For `randomness_account` Allows Fake VRF Injection

Location

gacha.rs:270

Description

In `gacha.rs`, the program accepts a `randomness_account` of type `UncheckedAccount<'info>` as input for the gacha pull process.

While the account is referenced and its data parsed in the `validate_vrf_for_purchase` and `get_vrf_randomness` helper functions, there is no explicit ownership check confirming that this account is actually owned by the Switchboard On-Demand program.

Because `UncheckedAccount` lacks ownership validation, an attacker could potentially pass in a malicious or fake VRF account that mimics the structure of a Switchboard randomness account.

This would allow the attacker to manipulate the randomness used for determining gacha outcomes, resulting in unfair or predictable rewards.

Recommendation

Explicitly validate the owner of the `randomness_account` to ensure it belongs to the Switchboard On-Demand program before any randomness is consumed.

Add a line similar to the following early in the `pull_gacha` and `reveal_gacha` handlers (or within your `validate_vrf_for_purchase` helper):

```
require!(
    ctx.accounts.randomness_account.owner ==
    SWITCHBOARD_ON_DEMAND_PROGRAM_ID.parse::<Pubkey>().unwrap(),
    GameError::InvalidVrfAccount
);
```

This ensures that:

- Only genuine Switchboard VRF accounts are accepted.
- The gacha results depend on cryptographically verifiable randomness.

Status

Fixed

High 02 `token_program` Can Be Any Program Implementing `TokenInterface` Leading To A Possibility Of Free Pulls

Location

gacha.rs

Description

In the `gacha.rs` :: `PullGacha` struct, there is a vulnerability due to insufficient validation of the `token_program` account. The code accepts any program that implements the `TokenInterface` trait:

```
pub token_program: Interface<'info, TokenInterface>,
```

Without explicitly verifying that the provided program is the official Token2022 program, a malicious user could:

1. Supply their own malicious program that implements `TokenInterface`
2. This program could pretend to burn tokens but actually not do it
3. The user would get gacha pulls without actually paying the WEED tokens

This could lead to economic exploitation of the game system, allowing users to perform gacha pulls without spending tokens.

Proof of Concept

Let's have the following scenario:

1. Attacker creates a malicious program implementing `TokenInterface`
2. When calling `pull_gacha`, attacker provides their malicious program as `token_program`
3. The `burn` CPI call goes to the malicious program which just returns success without burning tokens
4. Attacker gets gacha pulls while keeping their WEED tokens

```rust
// The vulnerable code that allows any token program
let burn_ctx = CpiContext::new(
    ctx.accounts.token_program.to_account_info(), // <-- No validation!
    burn_accounts,
);

burn(burn_ctx, total_cost)?;
```

Recommendation

Add an account constraint to validate that the token program matches the official Token2022 program Id:

```rust
use anchor_spl::token_interface::{TokenInterface, TokenAccount, Mint, Burn, burn, spl_token_2022};

// ... existing code ...

#[derive(Accounts)]
pub struct PullGacha<'info> {
    // ... existing accounts ...

    #[account(
        constraint = token_program.key() == spl_token_2022::ID @ GameError::InvalidTokenProgram
    )]
    pub token_program: Interface<'info, TokenInterface>,

    // ... existing accounts ...
}

// ... existing code ...
```

Make sure add `InvalidTokenProgram` to your error enum.

Status

Fixed

Medium

Mid 01 Poor VRF `seed_slot` Validations

Location

vrf.rs:30

vrf.rs:36

Description

In `vrf.rs` there is a logic error in the VRF validation code that incorrectly assumes `seed_slot` could be in the future. Looking at the actual code:

```
require!(
    seed_slot ⩾ current_slot.saturating_sub(1),
    GameError::InvalidVrfAccount
);

let future_slots = seed_slot.saturating_sub(current_slot);
require!(
    future_slots ⩽ MAX_SLOT_DIFFERENCE_STALE,
    GameError::VrfTooFarInFuture
);
```

The issue stems from a misunderstanding of how Switchboard VRF works:

1. The `seed_slot` is set when the VRF request is initially made;
2. The VRF proof is committed in a the same or later slot;
3. Therefore, `seed_slot` will always be less than or equal to the slot where the proof is committed;
4. This means:
    - The check for future slots (`future_slots ⩽ MAX_SLOT_DIFFERENCE_STALE`) is pointless;
    - The strict recency check (`seed_slot ⩾ current_slot.saturating_sub(1)`) is too aggressive.

Recommendation

Replace the current validation with a more appropriate check that ensures the VRF request isn't too old:

In `constants.rs`:

```
// ... existing code ...

/// Maximum allowed age (in slots) for a VRF request
/// Set to 5 slots to allow for normal network latency while
    preventing replay of old requests
pub const MAX_VRF_AGE_SLOTS: u64 = 5;
```

In `vrf.rs`:

```
let age_in_slots = current_slot.saturating_sub(seed_slot);
require!(
    age_in_slots ⩽ MAX_VRF_AGE_SLOTS,
    GameError::VrfTooOld
);
```

This change:

1. Removes the incorrect future slot check;
2. Replaces the overly strict recency check with a reasonable age limit;
3. Prevents replay of old VRF requests.

Status

Fixed

Mid 02 Loss Of User Funds When Switchboard VRF Fails Or Stalls

Location

gacha.rs:541

Description

In the current gacha flow, tokens are burned immediately when a user calls `pull_gacha`, before the Switchboard VRF oracle provides randomness.

If the VRF account becomes invalid or the oracle fails to respond (e.g., due to queue failure, network issue, or on-chain corruption), the user cannot complete the `reveal_gacha` step. Although `clear_expired_vrf` allows clearing the stuck request, it does not refund the burned tokens. As a result, users permanently lose tokens even though no randomness was produced and no gacha rewards were received.

Recommendation

Implement a refund mechanism for failed VRF requests.

- Refund on failure: Allow `clear_expired_vrf` to return tokens if the VRF was invalidated by Switchboard or never fulfilled.
- Escrow pattern: Hold tokens in an escrow account between pull and reveal to protect users against oracle-side failures.

Status

Fixed

An event is emitted when `clear_expired_vrf` gets called and the team will manually refund users that called it successfully.

Low

Low 01 Private

Description

The client has asked to keep this finding private.

Low 02 Incorrect Time Annotation For `MAX_VRF_AGE_SLOTS`

Location

gacha.rs:603

Description

The constant `MAX_VRF_AGE_SLOTS` is currently defined as:

```
const MAX_VRF_AGE_SLOTS: u64 = 5000; // ~40 minutes at 400ms slots
```

However, the comment inaccurately states that 5000 slots correspond to approximately 40 minutes. At a slot duration of 400 milliseconds, 5000 slots actually amount to around 33 minutes and 20 seconds, not 40 minutes.

Recommendation

Update the inline comment to reflect the accurate duration. For example:

```
const MAX_VRF_AGE_SLOTS: u64 = 5000; // ~33 minutes at 400ms slots
```

Alternatively, if 40 minutes is the intended target duration, consider adjusting the constant value to match it more closely:

```
40 minutes ÷ 0.4 seconds/slot = 6000 slots
```

So, to represent approximately 40 minutes, use:

```
const MAX_VRF_AGE_SLOTS: u64 = 6000; // ~40 minutes at 400ms slots
```

Status

Fixed

Low 03 Discrepancy About Weapons Probabilities Between Comments And Actual Implementation

Location

gacha.rs:60

gacha.rs:110

Description

In `gacha.rs` :: `initialize` there is a discrepancy between the documented and actual probabilities for rare weapons in the gacha system. The code comment states that rare weapons should have a 30% total probability:

```
// Rare (30%): 4 weapons  // <-- Comment claims 30%
```

However, when summing up the actual probabilities of the rare weapons (rarity = 2):

```
weapon_config.weapons[6] = WeaponData {
    name: format_weapon_name("Rare Street Sweeper")?,
    rarity: 2,
    probability: 1000, // 10%
};

weapon_config.weapons[7] = WeaponData {
    name: format_weapon_name("Rare Silenced 9")?,
    rarity: 2,
    probability: 600,  // 6%
};

weapon_config.weapons[8] = WeaponData {
    name: format_weapon_name("Rare MP5 SD")?,
    rarity: 2,
    probability: 600,  // 6%
};

weapon_config.weapons[9] = WeaponData {
    name: format_weapon_name("Rare AK-47 Chopper")?,
    rarity: 2,
    probability: 500,  // 5%
};
```

The actual sum is: 10% + 6% + 6% + 5% = 27%, which is 3% less than documented.

Recommendation

The total sum of all probabilities is 10,000 bps (100%) meaning that the current actual value of 27% is intended. Update the code comment to reflect the actual probability:

```
// Rare (27%): 4 weapons
```

Status

Fixed

Low 04 Non Unique Pull Id Generation

Location

gacha.rs:352

Description

The current implementation generates a pull Id using the system timestamp from Solana's `Clock` sysvar:

```
let pull_id = Clock::get()?.unix_timestamp as u64;
```

This approach relies solely on the current Unix timestamp, which is not guaranteed to be unique on the Solana blockchain. Multiple transactions can occur within the same slot (meaning at the same second) and thus produce identical timestamps. As a result, this may lead to Id collisions.

Recommendation

To ensure each pull Id is unique and collision-resistant, it is recommended to derive the Id using additional entropy or stateful sequencing:

Option 1 — Combine Timestamp with Slot and Hash

Mix timestamp, slot value and user entropy and hash them to produce a unique numeric Id:

```
let clock = Clock::get()?;

let mut entropy_data = Vec::new();
entropy_data.extend_from_slice(&clock.unix_timestamp.to_le_bytes());
entropy_data.extend_from_slice(&clock.slot.to_le_bytes());
entropy_data.extend_from_slice(ctx.accounts.user.key().as_ref());

let entropy_hash = hash::hash(&entropy_data);
let pull_id = u64::from_le_bytes(
    entropy_hash.to_bytes()[..8].try_into()
        .map_err(|_| GameError::CalculationOverflow)?
);
```

This method significantly reduces collision likelihood without requiring additional on-chain state.

Option 2 — Use a Stateful Counter

If persistent program state is available, store and increment a counter for each new pull:

```
pub struct PullState {
    pub counter: u64,
}

pull_state.counter += 1;
let pull_id = pull_state.counter;
```

This ensures deterministic, globally unique identifiers.

Option 3 — Use a PDA for Deterministic Uniqueness

If each pull needs a unique account, derive a Program Derived Address (PDA) using the user's public key and timestamp:

```
let seeds = &[
    b"pull",
    user.key.as_ref(),
    &Clock::get()?.unix_timestamp.to_le_bytes(),
];
```

```
let (pull_pda, _bump) = Pubkey::find_program_address(seeds, program_id
    );
```

Each derived address will be unique and collision-free within the program's namespace.

Status

Fixed

Low 05 Faulty Not Revealed Check Rejecting A Valid Randomness Case

Location

vrf.rs:66

Description

In the `vrf.rs::get_vrf_randomness` function, there's a check to verify if the VRF value has been revealed by checking if the randomness bytes are not all zeros:

```
let has_randomness = randomness.iter().any(|&b| b != 0);
if !has_randomness {
    return Err(GameError::VrfNotRevealedState.into());
}
```

This approach has a flaw - there is a theoretical possibility (1 in 2^256) that a valid revealed VRF value could consist of all zeros. In this extremely rare case, the function would incorrectly reject the valid randomness by throwing a `VrfNotRevealedState` error.

Recommendation

Replace the current zero-check with the Switchboard SDK's official method to verify if the value is revealed:

```rust
pub fn get_vrf_randomness(data: std::cell::Ref<&mut [u8]>) -> Result<[u8; 32]> {
    // Parse VRF account data
    let randomness_data = RandomnessAccountData::parse(data)
        .map_err(|_| GameError::InvalidVrfAccount)?;

    // Use SDK method to check if value is revealed
    let clock = Clock::get()?;
    match randomness_data.get_value(&clock) {
        Ok(value) => Ok(value),
        Err(_) => Err(GameError::VrfNotRevealedState.into())
    }
}
```

This approach is more robust as it uses the official SDK method to verify the VRF state, eliminating the edge case possibility.

Status

Fixed

Low 06 Redundant Queue Structure Of Pending Pulls

Location

gacha.rs

state.rs

Description

In the `gacha.rs::pull_gacha` function, there is a requirement that enforces the user's queue to be empty before making a new pull:

```rust
require!(
    user_queue.pending_pulls.is_empty(),
    GameError::QueueFull
);
```

This creates a design inconsistency because:

1. The code maintains a `Vec<PendingPull>` queue structure and a `max_queue_size` field in `UserGachaQueue`
2. However, users are forced to have an empty queue before making any new pulls
3. The `max_queue_size` field is initialized but never used for validation since only one pull at a time is allowed
4. This makes both the queue structure and the `max_queue_size` field redundant since the system effectively only allows one pending pull at a time

Recommendation

Choose one of these two approaches:

1. Remove the queue structure and simplify to a single pending pull:

```
#[account]
pub struct UserGachaQueue {
    pub owner: Pubkey,
    pub current_pull: Option<PendingPull>,
}
```

2. Or remove the empty queue requirement and actually use the queue structure:

```
// In pull_gacha()
require!(
    user_queue.pending_pulls.len() < user_queue.max_queue_size as
    usize,
    GameError::QueueFull
);
```

The first approach is recommended since the immediate reveal pattern is already enforced by the protocol design, making a queue structure unnecessary.

Status

Fixed

Informational

Info 01 `ProbabilitiesUpdatedEvent` Does Not Contain Updated And Old Values

Location

admin.rs

Description

The `update_weapon_probabilities` function emits a `ProbabilitiesUpdatedEvent` that lacks important information about the probability changes. Unlike the `PriceUpdatedEvent` which includes both `old_price` and `new_price`, the probability update event only contains the admin address and timestamp:

```
emit!(ProbabilitiesUpdatedEvent {
    admin: ctx.accounts.admin.key(),
    timestamp: Clock::get()?.unix_timestamp,
});
```

This makes it difficult for off-chain services to:

1. Track historical probability changes
2. Monitor specific weapon probability adjustments
3. Audit changes to the gacha system's odds
4. Detect potentially problematic probability modifications

Recommendation

Enhance the `ProbabilitiesUpdatedEvent` to include both old and new probabilities. This change will provide complete information about probability changes, enabling better monitoring and transparency of the gacha system.

Status

Fixed

Info 02 `MAX_VRF_AGE_SLOTS` Constant Should Be Moved To Constants File

Location

gacha.rs

Description

In `gacha.rs`, the `MAX_VRF_AGE_SLOTS` constant is defined directly within the `check_vrf_clearable` function:

```
const MAX_VRF_AGE_SLOTS: u64 = 5000; // ~40 minutes at 400ms slots
```

This is inconsistent with the project's pattern where other configuration constants are centralized in constants.rs. The constants.rs file already contains other VRF-related constants like `MIN_VRF_FEE_LAMPORTS` and `MAX_SLOT_DIFFERENCE_STALE` in a dedicated "VRF SECURITY PARAMETERS" section.

Recommendation

Move the `MAX_VRF_AGE_SLOTS` constant to the VRF security parameters section in `constants.rs`:

```
// ... existing code ...

pub const MAX_VRF_AGE_SLOTS: u64 = 5000;

// ... existing code ...
```

Then update the `check_vrf_clearable` function to use the imported constant:

```
use crate::constants::*;

// In check_vrf_clearable function:
if current_slot > randomness_data.seed_slot &&
   current_slot.saturating_sub(randomness_data.seed_slot) >
   MAX_VRF_AGE_SLOTS {
    // ... existing code ...
}
```

This change improves code organization and maintainability by keeping all configuration constants in a single location.

Status

Fixed

Info 03 Remove Redundant Parameter `check_vrf_claimable`

Location

gacha.rs:585

Description

In the function `check_vrf_claimable`, the parameter `pull_timestamp` is declared but never used within the function body. This makes it redundant and may lead to confusion for future maintenance.

Recommendation

Remove the unused `pull_timestamp` parameter from the function definition to simplify the code and eliminate unnecessary arguments.

Status

Fixed

Info 04 Use `RandomnessAccountData::parse` Instead Of Manual Byte Offsets In `get_vrf_randomness`

Location

vrf.rs:63

Description

The function `get_vrf_randomness` currently extracts randomness bytes directly from a hard-coded offset (`144 .. 176`) in the VRF account data. This approach manually assumes the internal byte layout of the Switchboard VRF account and bypasses the `RandomnessAccountData::parse` method provided by the Switchboard SDK.

Relying on fixed offsets makes the function fragile: if the SDK or account layout changes, the extraction logic could break or return incorrect randomness. Additionally, parsing errors and validation logic from the SDK are skipped, reducing data integrity guarantees.

Recommendation

Use the Switchboard SDK's `RandomnessAccountData::parse` function to safely parse the VRF account data and retrieve the randomness value.

Status

Fixed

Info 05 Missing Admin Transfer Functionality

Location

admin.rs

Description

`admin.rs` lacks functionality to change or rotate the admin address in the `WeaponConfig` account.

The admin address is set during initialization and used for critical operations like:

- Toggling the gacha system on/off
- Updating weapon probabilities

- Updating WEED token price per pull

However, there is no function to update this admin address. This creates a security risk because:

1. If the admin's private key is compromised, an attacker gains permanent control
2. There's no way to rotate keys as part of security best practices
3. If the admin key is lost, the system becomes permanently unmanageable

Recommendation

Add functionality to transfer admin control to another address. Consider adding a timelock to this function for security.

Status

Acknowledged