



RACERDOTFUN

Security Review



NOVEMBER, 2025

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

Lead Auditors



PeterSR



0x539.eth

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Commit Hash
 - Scope
 - Roles
- Executive Summary
 - Issues Found
 - Security Grade
- Findings
 - High
 - Low
 - Informational

Protocol Summary

RacerDotFun is a Solana-based protocol implemented as an Anchor program for managing a token vault in a racing game ecosystem. It handles token deposits into a shared vault, admin-registered payouts for race participants (based on points and amounts), referral bonuses, and claims for pending distributions. The vault supports a single mint (token type), uses PDA-derived accounts for security,

and includes admin controls for pausing, authority transfers, and reconciliation. The program is pausable for maintenance, and all transfers use SPL Token standards.

Disclaimer

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

Risk Classification

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

Audit Details

Commit Hash

`c86e5f5eed711c50eb4d2fab580f6152a249a941`

Scope

Id	Files in scope
1	lib.rs

Roles

Id	Roles
1	Admin
2	User

Executive Summary

Issues Found

Severity	Count	Description
High	2	Critical vulnerabilities
Medium	0	Significant risks
Low	7	Minor issues with low impact
Informational	2	Best practices or suggestions
Gas	0	Optimization opportunities

Security Grade



Based on the audit findings and code quality, the overall protocol security grade

is **83.00 / 100.00**. This reflects a medium security posture with some room for improvement.

Findings

High

High 01 **close** Function Permanently Locks Vault Funds

Location

lib.rs:245

Description

The `close` instruction closes the config account without verifying the vault is empty. This permanently locks all tokens in the vault because:

1. All claim functions require the config account to exist for validation
2. The `vault_signer` PDA (which owns the vault tokens) is derived using `config.key()`
3. After config is closed, no program instructions can access the vault. While the vault tokens technically still exist on-chain, they become permanently inaccessible through the program's interface.

```
1 pub fn close(ctx: Context<Close>) -> Result<()> {  
2     // No check if vault has tokens!  
3     emit!(ConfigCloseEvent {  
4         authority: ctx.accounts.authority.key(),  
5         timestamp: Clock::get()?.unix_timestamp,  
6     });  
7     Ok(())  
8 }
```

Proof of Concept

Scenario:

1. Vault holds 500,000 USDC
2. Admin calls `close()`
3. Config account is closed successfully
4. Tokens remain in `vault_token` account

The lock is permanent:

- `vault_signer` PDA requires `config.key()` to derive: `[b"vault_signer", config.key()]`
- All claim functions validate: `#[account(seeds = [b"config", mint.key()], bump)]`
- Without config, Anchor rejects all transactions
- 500,000 USDC locked forever (or until program is modified and redeployed)

Additional Impact:

- All registry accounts (payout registries, referrer registries, receipts) become orphaned
- Rent in these accounts is also locked
- Example: 1,000 registries × 0.001 SOL = 1 SOL locked in rent alone

Recommendation

Add a safety check to prevent closing when the vault contains tokens:

```
1  #[derive(Accounts)]
2  pub struct Close<'info> {
3      #[account(
4          mut,
5          seeds = [b"config", mint.key().as_ref()],
6          bump,
7          has_one = authority,
8          close = authority
9      )]
10     pub config: Account<'info, Config>,
11
12     #[account(
13         seeds = [b"vault_signer", config.key().as_ref()],
14         bump = config.vault_signer_bump
```

```

15     )]
16     /// CHECK: PDA signer
17     pub vault_signer: UncheckedAccount<'info>,
18
19     pub mint: Account<'info, Mint>,
20
21     // Add vault_token to verify it's empty
22     #[account(
23         associated_token::mint = mint,
24         associated_token::authority = vault_signer
25     )]
26     pub vault_token: Account<'info, TokenAccount>,
27
28     #[account(mut)]
29     pub authority: Signer<'info>,
30 }
31
32 pub fn close(ctx: Context<Close>) -> Result<()> {
33     // Verify vault is empty before closing
34     require!(
35         ctx.accounts.vault_token.amount == 0,
36         VaultError::VaultNotEmpty
37     );
38
39     emit!(ConfigCloseEvent {
40         authority: ctx.accounts.authority.key(),
41         timestamp: Clock::get()?.unix_timestamp,
42     });
43     Ok(())
44 }
45
46 // Add new error variant
47 #[error_code]
48 pub enum VaultError {
49     // ... existing errors
50     #[msg("Cannot close config while vault contains tokens")]
51     VaultNotEmpty,
52 }

```

Status

Acknowledged

High 02 Missing PDA Seeds Validation Allows Cross-config Vault Drainage

Location

lib.rs:128

Description

The `ClaimPendingPayouts` and `ClaimPendingBonuses` functions fail to validate that the registry accounts belong to the correct config instance. The constraints only check that the recipient/referrer field matches, but do not verify the PDA seeds include the current config's public key.

```
1 // ClaimPendingPayouts - VULNERABLE
2 #[account(
3     mut,
4     constraint = payout_registry.recipient == recipient.key()
5 )]
6 pub payout_registry: Account<'info, PayoutRegistry>,
7
8 // ClaimPendingBonuses - VULNERABLE
9 #[account(
10    mut,
11    constraint = referrer_registry.referrer == referrer.key()
12 )]
13 pub referrer_registry: Account<'info, ReferrerRegistry>,
```

This allows an attacker to use a `PayoutRegistry` or `ReferrerRegistry` from a different config to drain tokens from the current vault.

Proof of Concept

Scenario:

1. Two vaults exist:
 - Vault A (USDC) with 100,000 tokens
 - Vault B (worthless BOBCOIN)
2. Attacker registers a payout in Vault B:

- Calls `register_payout` on Config B
- Creates registry: `[b"payout_registry", configB.key(), attacker.key()]`
- Sets `total_pending = 100,000` BOBCOIN

3. Attacker exploits Vault A:

- Calls `claim_pending_payouts` with:
 - config: Config A (USDC vault)
 - payout_registry: Registry from Config B (with 100,000 pending)
 - recipient: Attacker's wallet

4. Validation passes:

- ✓ `payout_registry.recipient == recipient.key()` (`attacker == attacker`)
- ✓ Registry is owned by the program
- ✓ Registry deserializes correctly
- ✗ No check that registry belongs to Config A

5. Result:

- Code reads `registry.total_pending = 100,000`
- Transfers 100,000 USDC from Vault A to attacker
- Vault A is drained using a registry from Vault B

Recommendation

Add PDA seeds validation to enforce that registries belong to the current config:

```

1 // ClaimPendingPayouts - FIXED
2 #[account(
3     mut,
4     seeds = [b"payout_registry", config.key().as_ref(), recipient.key()
5             ().as_ref()],
6     bump,
7     constraint = payout_registry.recipient == recipient.key()
8 )]
9 pub payout_registry: Account<'info, PayoutRegistry>,
10
11 // ClaimPendingBonuses - FIXED
12 #[account(
13     mut,
14     seeds = [b"referrer_registry", config.key().as_ref(), referrer.key()
15             ().as_ref()],

```

```
14     bump,  
15     constraint = referrer_registry.referrer == referrer.key()  
16 ]]  
17 pub referrer_registry: Account<'info, ReferrerRegistry>,
```

The `seeds` constraint forces Anchor to validate the account is at the expected PDA address, preventing cross-config attacks.

Status

Acknowledged

Low

Low 01 Missing Tests For The Whole Protocol

Location

lib.rs

Description

The codebase has many stateful flows, cross-account invariants, PDAs, CPIs, arithmetic operations, and permission checks that are only validated by code review. Without automated tests, regressions or subtle logic errors can be introduced and remain undetected.

Because this is a financial/token program, even low-severity logic regressions can cause funds to be locked, mis-accounted, or cause DoS for users.

Recommendation

Add a comprehensive automated test suite and CI to exercise normal, boundary, and adversarial behaviors. Minimum recommended actions:

- Unit tests;
- Integration tests;

- Fuzz tests;
- Continuous Integration.

Implementing the above will materially reduce regression risk and give confidence that token flows and permission rules behave as intended.

Status

Acknowledged

Low 02 Module Separation Missing

Location

`lib.rs`

Description

The protocol is implemented as a single large `lib.rs` file. That increases maintenance and review burden: related concerns (instruction handlers, account/context structs, state types, events, and errors) are mixed together, which makes the code harder to navigate, reason about, and test. For reviewers and contributors this leads to slower audits, higher chance of subtle mistakes, and lower developer velocity. It also makes it harder to write focused unit tests, reuse helpers, and apply API-level changes safely.

Recommendation

Refactor into logical modules and smaller files to separate concerns and improve reviewability, testability, and reuse.

Split `lib.rs` into modules, for example:

- `lib.rs` — top-level mod declarations and re-exports;
- `programs/src/instructions/*.rs` — one file per instruction group (`initialize.rs`, `payout.rs`, `referral.rs`, `claim.rs`, `admin.rs`);
- `programs/src/accounts.rs` — all Anchor `#[derive(Accounts)]` context structs;

- `programs/src/state.rs` — account data structs (Config, Registry, Receipt, etc.) and size constants;
- `programs/src/events.rs` — event structs;
- `programs/src/errors.rs` — `VaultError` enum and messages;
- `programs/src/utils.rs` — shared helpers.

This refactor reduces review complexity, makes audits more effective, and lowers the chance of logic regressions.

Status

Acknowledged

Low 03 Deposit Function Bypasses Pause Mechanism

Location

lib.rs:32

Description

All administrative functions (`register_payout`, `register_referral_bonus`, `claim_pending_payouts`, `claim_pending_bonuses`) check if the program is paused before executing:

```
1 require!(!config.paused, VaultError::ProgramPaused);
```

However, the `deposit` function does not include this check, allowing users to deposit tokens even when the program is paused for emergency maintenance.

```
1 pub fn deposit(ctx: Context<Deposit>, amount: u64) -> Result<> {
2     require!(amount > 0, VaultError::ZeroAmount);
3     // Missing: require!(!ctx.accounts.config.paused, VaultError::
4         ProgramPaused);
5
6     let cpi_accounts = Transfer {
7         from: ctx.accounts.depositor_token.to_account_info(),
8         to: ctx.accounts.vault_token.to_account_info(),
```

```
8         authority: ctx.accounts.depositor.to_account_info(),
9     };
10    // ...
11 }
```

Recommendation

Option 1: Add pause check to deposit

If deposits should be blocked during emergencies:

```
1 pub fn deposit(ctx: Context<Deposit>, amount: u64) -> Result<()> {
2     require!(amount > 0, VaultError::ZeroAmount);
3     require!(!ctx.accounts.config.paused, VaultError::ProgramPaused);
4
5     let cpi_accounts = Transfer {
6         from: ctx.accounts.depositor_token.to_account_info(),
7         to: ctx.accounts.vault_token.to_account_info(),
8         authority: ctx.accounts.depositor.to_account_info(),
9     };
10    // ...
11 }
```

Option 2: Document intentional behavior

If deposits should be allowed during pause (to ensure users can always add funds):

```
1 pub fn deposit(ctx: Context<Deposit>, amount: u64) -> Result<()> {
2     require!(amount > 0, VaultError::ZeroAmount);
3     // Note: Deposits allowed even when paused to ensure users can add
4     // funds
5
6     let cpi_accounts = Transfer {
7         // ...
8     };
9     // ...
10 }
```

Status

Acknowledged

Low 04 Authority Transfer Lacks Safety Checks

Location

lib.rs:206

Description

The `transfer_authority` function allows the current authority to transfer control to a new address without any validation. If the new authority is set to an invalid address (PDA without known seeds, wrong address due to typo, or non-signer account), admin access is permanently lost.

```
1 pub fn transfer_authority(  
2     ctx: Context<TransferAuthority>,  
3     new_authority: Pubkey,  
4 ) -> Result<()> {  
5     let config = &mut ctx.accounts.config;  
6     let old_authority = config.authority;  
7     config.authority = new_authority; // No validation!  
8  
9     emit!(AuthorityTransferEvent {  
10         old_authority,  
11         new_authority,  
12         timestamp: Clock::get()?.unix_timestamp,  
13     });  
14  
15     Ok(())  
16 }
```

Recommendation

Implement a propose-accept pattern like OpenZeppelin's Ownable2Step:

```
1 #[account]  
2 pub struct Config {  
3     pub authority: Pubkey,  
4     pub pending_authority: Option<Pubkey>, // Add this field  
5     pub mint: Pubkey,  
6     pub vault_signer_bump: u8,  
7     pub paused: bool,  
8 }  
9
```

```

10 pub fn transfer_authority(
11     ctx: Context<TransferAuthority>,
12     new_authority: Pubkey,
13 ) -> Result<()> {
14     let config = &mut ctx.accounts.config;
15     config.pending_authority = Some(new_authority);
16
17     emit!(AuthorityTransferProposedEvent {
18         current_authority: config.authority,
19         pending_authority: new_authority,
20         timestamp: Clock::get()?.unix_timestamp,
21     });
22
23     Ok(())
24 }
25
26 pub fn accept_authority(ctx: Context<AcceptAuthority>) -> Result<()> {
27     let config = &mut ctx.accounts.config;
28
29     require!(
30         config.pending_authority == Some(ctx.accounts.new_authority.
31         key()),
32         VaultError::NotPendingAuthority
33     );
34
35     let old_authority = config.authority;
36     config.authority = ctx.accounts.new_authority.key();
37     config.pending_authority = None;
38
39     emit!(AuthorityTransferredEvent {
40         old_authority,
41         new_authority: config.authority,
42         timestamp: Clock::get()?.unix_timestamp,
43     });
44
45     Ok(())
46 }
47
48 #[derive(Accounts)]
49 pub struct AcceptAuthority<'info> {
50     #[account(
51         mut,
52         seeds = [b"config", mint.key().as_ref()],
53         bump
54     )]
55     pub config: Account<'info, Config>,

```

```
56     pub mint: Account<'info, Mint>,  
57  
58     // New authority must sign to accept  
59     pub new_authority: Signer<'info>,  
60 }
```

Status

Acknowledged

Low 05 Admin Can Register Any Amount They Want Through `register_payout` And `register_referral_bonus`

Location

lib.rs

Description

The `register_payout` and `register_referral_bonus` functions in the code allow the admin (authority) to register payouts and bonuses for any recipient, referrer, or referee. This means that the admin can specify arbitrary amounts, points, and race IDs, provided that the `race_id_hash` matches the provided `race_id` in the `register_payout` function. This flexibility poses a significant security risk, as it could lead to abuse by the admin. For instance, an admin could potentially register excessive payouts or bonuses to themselves or collude with others to exploit the system, resulting in financial losses or unfair advantages.

Recommendation

To mitigate this vulnerability, it is recommended to implement stricter validation and constraints on the `register_payout` and `register_referral_bonus` functions. Consider introducing limits on the amounts and points that can be registered, as well as requiring additional checks to ensure that the recipient, referrer, and referee are legitimate and authorized entities. This would enhance the overall security and integrity of the payout and referral bonus system.

Status

Acknowledged

Low 06 Missing `race_id` Length Validation In `register_payout`

Location

lib.rs:65

Description

The code for the `register_payout` function lacks a validation check to ensure that the length of the `race_id` string does not exceed 32 bytes. This is an oversight, as the `race_id` is used as a key component in the program's logic and is expected to conform to certain constraints. This issue is concerning given that a similar validation is present in the `register_referral_bonus` function, highlighting an inconsistency in the codebase.

Recommendation

To address this vulnerability, it is recommended to implement a validation check in the `register_payout` function to ensure that the length of the `race_id` does not exceed 32 bytes. This can be done by adding the following line of code:

```
1 require!(race_id.len() ≤ 32, VaultError::RaceIdTooLong);
```

This validation should be placed before any logic that relies on the `race_id`, ensuring that only valid inputs are processed.

Status

Acknowledged

Low 07 Missing Validation Of Registered Amounts

Location

lib.rs

Description

The code contains a vulnerability in the `register_payout` and `register_referral_bonus` functions where there is no validation to ensure that the amount being registered does not exceed the current balance of the vault token. This oversight allows a malicious admin (or a hacked admin key) to register payouts or bonuses that exceed the actual available balance in the vault. As a result, this could lead to situations where the system appears to have more funds available for payouts than it actually does, causing financial discrepancies and undermining the integrity of the payout system.

Recommendation

To mitigate this vulnerability, it is recommended to implement checks in both the `register_payout` and `register_referral_bonus` functions to ensure that the amount being registered do not exceed the current `vault_token.amount`. This can be done by adding a `require!` statement that validates the amount before it is added to the total pending amounts.

For example:

```
1 require!(  
2     ctx.accounts.vault_token.amount ≥ global_registry.total_pending +  
3     amount,  
4     VaultError::InsufficientBalance  
5 );
```

Status

Acknowledged

Informational

Info 01 `PayoutAlreadyExists` Is Never Used

Location

`lib.rs`

Description

The code contains an error variant `PayoutAlreadyExists` that is defined but never utilized within the program's logic. This oversight means that when a duplicate registration attempt occurs, the program will not return a user-friendly error message specific to this situation. Instead, it will produce a generic Anchor runtime error indicating that the account already exists. This can lead to confusion for users, as they may not understand the reason for the failure, making it difficult to troubleshoot issues related to duplicate registrations.

Recommendation

To improve user experience and clarity, it is recommended to implement the `PayoutAlreadyExists` error variant in the relevant sections of the code where duplicate payout registrations are checked. This will allow the program to return a specific error message when such an attempt is made, providing users with clearer feedback on the nature of the error and guiding them on how to resolve it.

Status

Acknowledged

Info 02 Redundant `claimed` Field In `ReferralBonus` Struct

Location

`lib.rs`

Description

The code contains a variable `claimed` within the `ReferralBonus` struct that is intended to track whether a referral bonus has been claimed. However, this variable is never set to `true` during the claim process, meaning that the system does not update its state to reflect that the bonus has been claimed. As a result, the `claimed` variable remains `false` indefinitely, which can lead to confusion.

Recommendation

To address this vulnerability, it is recommended to remove the `claimed` variable from the `ReferralBonus` struct as it is not used and not needed

Example code modification:

```
1 pub struct ReferralBonus {  
2     pub race_id: String,  
3     pub referrer: Pubkey,  
4     pub referee: Pubkey,  
5     pub amount: u64,  
6     pub timestamp: i64,  
7 }
```

Status

Acknowledged