Relatório de Auditoria de Contrato Inteligente: Braza Token (Revisão Refinada)

Auditor: Soroban Auditor Pro (Manus AI) **Data:** 13 de Dezembro de 2025 **Contrato Auditado:** `braza-token` (Repositório: `https://github.com/LelloTereciani/braza`) **Escopo:** Código-fonte (ignorado `README.md` desatualizado) **Versão do SDK:** `soroban-sdk = "21.0.0"`

---

1. Visão Geral e Alinhamento com Soroban

O contrato `braza-token` é um Stellar Asset Contract (SAC) que implementa funcionalidades de token padrão (SEP-0041) e adiciona um sofisticado sistema de **Vesting** baseado em ledgers, além de mecanismos de **Administração** (mint/burn) e **Conformidade** (pause/blacklist).

A análise do código-fonte confirma que o projeto está alinhado com as melhores práticas do Soroban SDK, utilizando:

- **Aritmética Segura:** Uso consistente de `checked_add` e `checked_sub` para prevenir overflows e underflows em operações de saldo e supply.
- **Padrão CEI (Checks-Effects-Interactions):** Implementação explícita do padrão CEI nas funções críticas (`transfer`, `transfer_from`, `mint`, `burn`, etc.), o que é fundamental para prevenir a maioria dos ataques de reentrância.
- **Gerenciamento de Storage (TTL):** Uso de `extend_ttl` (`bump`) para garantir a persistência do storage crítico (`CRITICAL_STORAGE_TTL`) e dos saldos.

---

2. Issues Identificados (Baseado Apenas no Código)

Os issues identificados são de severidade Média e Baixa, focados em otimização de budget, consistência de design e boas práticas de desenvolvimento em Rust/Soroban.

ID	Descrição do Problema	Severidade	Localização
2.1	**Ausência de Bump em `approve`**	Média	`token.rs`, `approve`
2.2	**Uso Inconsistente do Reentrancy Guard**	Média	`token.rs` (várias funções)

2.3	**Uso de `unwrap()` em Funções Críticas de Storage**	Baixa	`storage.rs` (linhas 99, 155)
2.4	**Nomenclatura de Variáveis em Português**	Baixa	Vários arquivos

Issue 2.1: Ausência de Bump em `approve` (Média)

Descrição: A função `approve` define um novo valor de allowance, mas falha em estender o Time-to-Live (TTL) do storage de allowance. O allowance é armazenado no storage persistente e, sem um `bump` explícito, pode expirar e ser removido do ledger, resultando em perda de estado e frustração do usuário.

Impacto: Perda inesperada de aprovações delegadas.

Correção Sugerida: Adicionar a chamada para `storage::bump_allowance` após a atualização do allowance.

Rust

```
// Em token.rs, dentro de pub fn approve(...):
// ...
// storage::set_allowance(&env, &from, &spender, amount);
//
// // ✅ CORREÇÃO: Fazer bump do TTL do allowance
// storage::bump_allowance(&env, &from, &spender); // <--- ADICIONAR ESTA
LINHA
//
// events::emit_approve(&env, &from, &spender, amount);
// ...
```

Issue 2.2: Uso Inconsistente do Reentrancy Guard (Média)

Descrição: O contrato utiliza um `REENTRANCY GUARD` armazenado no storage de instância (`env.storage().instance()`) para proteger funções que não realizam chamadas externas (CCCs). No Soroban, o padrão CEI é a principal defesa contra reentrância em CCCs. Para funções que apenas manipulam o estado interno, o guard é desnecessário e consome budget de CPU e Storage em cada transação.

Impacto: Budget de CPU/Storage desnecessariamente consumido; design confuso.

Correção Sugerida: Remover o `REENTRANCY GUARD` de todas as funções que não realizam chamadas externas. Se a intenção for proteger contra reentrância em futuras CCCs, o guard

deve ser implementado usando o storage temporário (`env.storage().temporary()`) para garantir que o lock seja limpo após a transação.

Issue 2.3: Uso de `unwrap()` em Funções Críticas de Storage (Baixa)

Descrição: Funções de leitura de storage de instância, como `get_admin` e `get_metadata` em `storage.rs` , utilizam `unwrap()` . Embora o contrato seja inicializado para garantir a presença desses valores, o uso de `unwrap()` pode levar a um `panic` em tempo de execução se o estado for inesperadamente ausente.

Impacto: Falha catastrófica (panic) em caso de estado ausente.

Correção Sugerida: Substituir `unwrap()` por `expect()` com uma mensagem de erro clara, ou retornar um `Result` e tratar o erro no nível da função do contrato.

Rust

```rust
// Em storage.rs, linha 99:
pub fn get_admin(env: &Env) -> Address {
    env.storage().instance().get(&ADMIN).expect("Admin not set. Contract not initialized.")
}
```

---

3. Análise de Funcionalidades Avançadas

3.1. Sistema de Vesting (Vesting Schedules)

O sistema de vesting é um ponto forte do contrato, demonstrando atenção à segurança e ao budget.

Recurso	Implementação	Avaliação
Base de Tempo	Usa `env.ledger().sequence()`	**Excelente.** Corrige o problema de dependência de `timestamp` (que é manipulável) em favor da sequência de ledger, mais segura para lógica de tempo.
Proteção DoS	Limites por beneficiário (`MAX_VESTING_SCHEDULES`), limite global (`MAX_GLOBAL_VESTING_SCHEDULES`), `VESTING_CREATION_COOLDOWN_`	**Excelente.** Mitiga o risco de esgotamento de budget de storage por criação excessiva de schedules.

	`LEDGERS` e `VESTING_STORAGE_FEE`.	
Aritmética	Uso de `saturating_sub` e `checked_mul` / `checked_div` em `calculate_vested_amount`.	**Excelente.** Garante que os cálculos de vesting sejam seguros e não causem pânico.

3.2. Gerenciamento de Storage (TTL)

O contrato implementa um gerenciamento de TTL robusto:

- **Storage Crítico:** `CRITICAL_STORAGE_TTL` (1 ano) e `CRITICAL_STORAGE_THRESHOLD` (30 dias) são usados para dados globais (admin, supply, metadata).
- **Saldos:** `bump_balance` é chamado em `transfer` e `transfer_from`, garantindo que saldos ativos não expirem.

Recomendação de Budget: A chamada a `storage::bump_critical_storage(&env)` é feita em todas as funções de leitura (ex: `name`, `symbol`, `total_supply`). Embora isso mantenha o contrato ativo, pode ser uma otimização excessiva para funções de leitura frequentes. Recomenda-se que o `bump` seja feito apenas nas funções de escrita ou em uma função de manutenção periódica, para economizar budget de CPU em cada consulta de leitura.

---

4. Pontuação de Segurança

O código-fonte demonstra um alto nível de segurança e maturidade, com proteções explícitas contra os vetores de ataque mais comuns em contratos inteligentes.

Pontuação Refinada: 9/10

A dedução de 1 ponto é mantida devido aos issues de design (Reentrancy Guard) e à falha na gestão de TTL do allowance (Issue 2.1), que, embora não sejam falhas críticas de perda de fundos, representam riscos de usabilidade e perda de estado.

---

5. Próximos Passos

Ações de Prioridade:

1. **Corrigir Issue 2.1 (Allowance Bump):** Implementar `storage::bump_allowance` na função `approve`.
2. **Revisar Issue 2.2 (Reentrancy Guard):** Remover o guard das funções internas para otimizar o budget.

Recomendações de Teste:

- **Testes de Expiração:** Adicionar testes que simulem a expiração do TTL de um allowance não utilizado para confirmar que a correção do Issue 2.1 funciona.
- **Testes de Budget:** Medir o consumo de budget de CPU e Storage antes e depois da remoção do Reentrancy Guard (Issue 2.2) para quantificar a otimização.

Referências

[1] Stellar Development Foundation. Stellar Asset Contract (SAC) Specification (SEP-0041). Disponível em:

[2] Stellar Development Foundation. Soroban Smart Contracts Development Guide. Disponível em:

[3] Soroban Docs. Storage and TTL. Disponível em: