Security Assessment Report

The ChampCoin (TCC)

This document proves, from the given source code alone, that The ChampCoin satisfies a set of concrete security properties. The arguments below are constructive and refer directly to function bodies and state transitions. No external libraries, proxies, upgrade hooks, or unverified code exist in scope.

Contract address: 0x1831257D6FEF2b83354a75b19B8aAf8f6514D3DA

Explorer Link: https://bscscan.com/token/0x1831257D6FEF2b83354a75b19B8aAf8f6514D3DA

Token Name : The ChampCoin

symbol : TCC

decimals : 18 decimals

Token Logo: 

Total supply: 70,000,000 TCC created once at deployment and sent to the deployer's address.

Type : Standard ERC-20/BEP-20 token.

Source Code Status: Verified ✅

SECURITY SUMMARY:

The ChampCoin (TCC) contract is intentionally simple and locked down: fixed supply, no fees, no freeze, and no mint/burn after deployment. From a contract-level perspective, that's a solid, low-risk design for everyday holders. However beware of similar fake tokens, phishing, and avoid approvals from untrusted third party apps.

GENERAL CAPABILITIES:

What it can and can’t do

It CAN:

- Let people send and receive TCC.
- Let you approve another address (like a DEX) to spend your TCC on your behalf.

It CANNOT (by design):

- **Mint** new tokens (no inflation).
- **Burn** tokens (supply won’t go down via the contract).
- **Freeze/blacklist** wallets.
- **Charge transfer fees.**
- **Pause** transfers.
- Make outside calls during transfers (reduces attack surface).

WHO CONTROLS WHAT?

The owner **cannot** mint, burn, freeze, skim fees, or change balances.

The owner can also **renounce ownership** (set it to zero address), which removes even that limited control.

RISK PROFILE (VERY LOW)

- **Fixed supply:** The contract code only creates tokens once. There’s no code path to create more later.
- **No external calls** in transfers/approvals: this cuts off common re-entrancy tricks.
- **Checked math:** The Solidity version used automatically blocks overflows/underflows.
- **Zero-address blocks:** The contract rejects sending to or approving the zero address (prevents accidental loss and nonsense approvals).

Simple FAQ

Q: Can the team print more TCC later?
A: **No.** The code doesn't have any mint function available after launch.

Q: Can the team freeze my wallet or charge transfer fees?
A: **No.** There are no such functions.

Q: Is there a burn feature?
A: **No.** Sending to the zero address is blocked and there's no `burn()` function.

Q: What happens if I approve a DEX for 1,000 TCC?
A: That DEX (or its smart contracts) can move up to 1,000 TCC from your wallet. Reduce or revoke later if you want.

TECHNICAL REPORT

---

0) Model & Notation

- **State.**
  `_totalSupply : uint256`
  `_balances : mapping(address ⇒ uint256)`
  `_allowances : mapping((address, address) ⇒ uint256)`
  `_owner : address`, `_pendingOwner : address`

- **Events** do not affect state.

- **Arithmetic.** Solidity ≥0.8.0 reverts on over/underflow except inside `unchecked` blocks.

- **Codebase.** Single contract; `_mint` is `internal` and invoked only once in the constructor.

We use Hoare-style reasoning `{Pre} f(...) {Post}` and inductive invariants over executions.

---

1) Supply-Cap Correctness

Theorem 1 (Fixed Upper Bound).

For all reachable states after construction, `totalSupply() == MAX_SUPPLY` and never exceeds `MAX_SUPPLY`.

Proof.

1. The constructor sets `_owner = msg.sender` and calls `_mint(_owner, MAX_SUPPLY)`.

2. `_mint` requires `to != 0` and computes `newSupply = _totalSupply + value`. It enforces `require(newSupply <= MAX_SUPPLY, "Cap exceeded")`. Initially `_totalSupply = 0`, so `newSupply = MAX_SUPPLY` and the call succeeds, setting `_totalSupply = MAX_SUPPLY`.

3. `_mint` is `internal` and **only called in the constructor**; there is no other caller in the code.

4. No function decreases `_totalSupply`. No other function increases `_totalSupply`. Therefore `totalSupply()` is exactly `MAX_SUPPLY` in every post-construction state and can never exceed it. ∎

---

2) Conservation of Tokens

Theorem 2 (Balance Conservation).

For any call to `transfer` or `transferFrom` that does not revert, the sum of balances remains constant and equals `MAX_SUPPLY`.

Proof.

- `_transfer(from, to, value)` is the only code path that changes balances post-construction. It requires `from != 0`, `to != 0`, `fromBal = _balances[from]`, and `fromBal >= value`. Inside one guarded `unchecked` block, it sets

`_balances[from] = fromBal - value`, then `_balances[to] += value`.

- No other storage slot is modified. Thus `Δ_balances[from] = -value` and `Δ_balances[to] = +value`, so the sum is unchanged.
- By Theorem 1, `sum(balances) = totalSupply = MAX_SUPPLY` after construction; conservation under `_transfer` preserves this invariant for all future states. ∎

---

3) Zero-Address Safety

Theorem 3 (No Balance or Allowance for `address(0)` is Creatable Post-Construction).

No successful external call can create a balance or allowance for the zero address, nor transfer to or from it.

Proof.

- `_transfer` reverts if `from == 0` **or** `to == 0`.
- `_approve` reverts if `tokenOwner == 0` **or** `spender == 0`.
- `_mint` reverts if `to == 0`; it is only called in the constructor, where `to = _owner != 0`.
  Hence, after deployment, no valid execution can produce non-zero `_balances[0]` or `_allowances[*,0]`/`_allowances[0,*]`, nor can it transfer to/from zero. ∎

---

4) Allowance Accounting & Safety

Lemma 4.1 (Monotone Decrement in `transferFrom`).

If `transferFrom(from,to,value)` succeeds, then the pre-state has `currentAllowance = _allowances[from][msg.sender] ≥ value`, and the post-state sets `_allowances[from][msg.sender] = currentAllowance - value`.

Proof.
Function body:

- Reads `currentAllowance = _allowances[from][msg.sender]`.
- If `currentAllowance < value` → revert.
- In an `unchecked` block, calls `_approve(from, msg.sender, currentAllowance - value)`.
- Then calls `_transfer(from, to, value)`.
  Thus the allowance is decreased exactly by `value` upon success. The decrement is safe because the guard prevents underflow. ∎

Lemma 4.2 (Approval Writes Are Exact).

`approve(spender, value)` and `_approve(tokenOwner, spender, value)` set `_allowances[tokenOwner][spender] = value` and emit `Approval` with that exact value.

Proof.
From `_approve`: assignment is direct; there are no additional arithmetic operations. ∎

Theorem 4 (Allowance Correctness).

For any `(owner,spender)`, `_allowances[owner][spender]` equals the last explicit `_approve(owner,spender,·)` value minus the sum of successful `transferFrom` spends by `spender` against `owner` since that approval (and never negative).

Proof.
By Lemma 4.2, approvals write exact values. By Lemma 4.1, each successful spend subtracts exactly the spent amount; guards prevent underflow. No other path mutates that mapping. ∎

---

5) Reentrancy & External-Call Surface

Theorem 5 (No Reentrancy from Token Operations).

`transfer`, `transferFrom`, `approve`, `increaseAllowance`, `decreaseAllowance` contain **no** external calls and thus cannot yield control to untrusted code mid-update.

Proof.
Inspection of bodies shows only: storage updates, arithmetic, and `emit` statements. No `call`, `delegatecall`, `staticcall`, `transfer` (ETH), or interface invocations appear. Solidity event emission is not an external call and does not transfer control. Therefore, no reentrancy vector exists along these code paths. ∎

---

6) Ownership Safety & Progress

Lemma 6.1 (Only Owner Can Initiate Transfer).

`transferOwnership(newOwner)` is gated by `onlyOwner`; it reverts if `msg.sender != _owner`.

Lemma 6.2 (Unique Acceptance Right).

`acceptOwnership()` requires `msg.sender == _pendingOwner` else reverts, then sets `_owner = _pendingOwner` and `_pendingOwner = 0`.

Lemma 6.3 (Renounce Clears Roles).

`renounceOwnership()` (gated by `onlyOwner`) sets `_owner = 0` and `_pendingOwner = 0`.

Theorem 6 (Two-Step Ownership Safety & Liveness).

- **Safety:** Ownership can only change to an address that previously became `_pendingOwner` via an owner-authorized call; no other address can seize ownership.

- **Liveness:** Once `transferOwnership(X)` succeeds, `X` can acquire ownership by calling `acceptOwnership()`; no other state transitions can prevent this, and the original owner may overwrite `_pendingOwner` by re-invoking `transferOwnership(Y)` if needed.

Proof.
Safety follows from Lemma 6.1 and Lemma 6.2: the only state that assigns `_owner` is inside `acceptOwnership`, guarded by identity equality to `_pendingOwner`. Liveness follows because no function can set `_pendingOwner` except `transferOwnership`, and acceptance

is always available to `_pendingOwner` until overwritten or renounced; no third-party action can interpose. ∎

---

7) Non-Mintable, Non-Burnable, Non-Pausable, Non-Fee Properties

Theorem 7 (No Post-Deploy Minting).

Post-construction, `_mint` is never callable (it is `internal` and referenced only in the constructor). No other function increases `_totalSupply` or any `_balances[*]` except via `_transfer`, which preserves the supply (Theorem 2). ∎

Theorem 8 (No Burning).

There is no `burn` function; `_transfer` prohibits `to == 0`. Therefore, no burn path exists. ∎

Theorem 9 (No Pausing/Blacklisting/Fees).

No state or function references any pause flag, blacklist set, fee variables, or redirection logic. Transfers do not alter amounts except the exact sender-to-recipient move. Thus there can be no fee skims, freezes, or selective blocks derivable from the code. ∎

---

8) Overflow/Underflow Soundness

Theorem 10 (No Arithmetic Wraparound in Checked Regions).

In all non-`unchecked` contexts, Solidity ^0.8.30 reverts on overflow/underflow, preventing wraparound.

Proof. Language semantics. ∎

Theorem 11 (Correctness of `unchecked` Blocks).

All `unchecked` arithmetic operations are preceded by guards that make the operation safe:

- In `_transfer`, `fromBal >= value` ensures `fromBal - value` does not underflow.

- In `transferFrom`, `currentAllowance >= value` ensures `currentAllowance - value` does not underflow.

No `unchecked` additions exist. Therefore, no silent wraparound is possible. ∎

---

9) Event Correctness

Theorem 12 (Standards-Consistent Events).

- `_approve` emits `Approval(owner, spender, value)` exactly matching the written allowance.
- `_transfer` emits `Transfer(from, to, value)` exactly matching the state deltas.
- Constructor mint emits `Transfer(0, _owner, MAX_SUPPLY)` as required by ERC-20 mint semantics.
- Ownership transitions emit `OwnershipTransferStarted` and `OwnershipTransferred` reflecting the actual role changes.

Proof. Direct reading of emit sites; arguments are the same variables used in state updates. ∎

---

10) BEP-20 Surface Completeness

Theorem 13 (Explorer Compatibility).

`getOwner()` returns `_owner`. This satisfies the BEP-20 explorer convention and is read-only; no side effects exist.

Proof. Pure view function returning the storage slot. ∎

---

11) Composability Guarantees

Theorem 14 (ERC-20 Method Set & Return Values).

The contract exposes `totalSupply`, `balanceOf`, `allowance`, `transfer`, `approve`, `transferFrom`, each returning `bool` where applicable, and emitting canonical events. Therefore, any integration expecting a vanilla ERC-20 will execute without adaptation.

Proof. Function signatures and return paths match the standard; there are no revert-on-success or nonstandard behaviors. ∎

---

12) Global Invariant Set

By induction on the execution trace (constructor base case; step case via §§2–11), the following invariants hold in **all** reachable post-construction states:

1. `totalSupply() == MAX_SUPPLY`.
2. `∀a: _balances[a] ≥ 0` (always true for `uint256`) and `∑_a _balances[a] == MAX_SUPPLY`.
3. `∀(o,s): _allowances[o][s]` equals the last approved value minus the sum of successful spends by `s` from `o` since that approval; it is never negative.
4. No successful call can read or write a balance or allowance for `address(0)`; no transfer to or from `address(0)` can succeed.
5. No external calls occur in state-mutating token functions; reentrancy into those functions is impossible.
6. Ownership can change only via the two-step protocol; no unauthorized party can gain ownership; renounce sets both owner and pending owner to zero.
7. There is no mechanism to mint, burn, pause, blacklist, or charge transfer fees.

All invariants are preserved by every public/external function and by every internal helper reachable after deployment.

---

13) Conclusions (Proved Properties)

- **Fixed supply** and **supply-cap safety** are guaranteed.
- **Token conservation** across transfers is guaranteed.
- **Allowance accounting** is exact and safe from arithmetic errors; the ERC-20 race condition is a property of the standard's semantics, not of this implementation's safety (no unauthorized spend is possible).
- **No reentrancy vector** exists in token flows due to the total absence of external calls.
- **Zero-address misuse** is structurally impossible post-deployment.
- **No privileged monetary actions** (mint/burn/fees/blacklists/pauses) are present or derivable.

Accordingly, with respect to the provided source and under standard EVM semantics for Solidity ^0.8.30, **The ChampCoin (TCC) fulfills the above security properties by construction.**