## Unexpected Behavior from Non-Compliant or “Weird” ERC20 Tokens

---

The ERC20 standard (EIP-20) is widely used but **loosely defined**. Many tokens in the wild:

- Deviate from the standard,
- Behave inconsistently (e.g., don’t return a boolean),
- Or have edge-case behaviors (e.g., non-reverting failures, changing balances arbitrarily).

If your contract assumes **strict ERC20 compliance**, it can silently break or be exploited when interacting with these “weird” tokens.

---

Types of Weird ERC20 Tokens

Here’s a categorized list of common misbehaviors seen in ERC20 tokens:

| Type | Description & Example | Risk |
|---|---|---|
| **No Return Value** | `transfer()` returns nothing (e.g., USDT) | Can skip failure checks |
| **Returns `true` on failure** | Always returns `true`, even if transfer fails | Logic proceeds incorrectly |
| **Reverts silently** | Reverts without a useful error message | Hard to debug, breaks flows |
| **Transfers less than asked** | Sends less than `_amount` without reverting | Accounting bugs |
| **Burns on Transfer** | Charges fee or burns tokens (e.g., deflationary) | Wrong balance calculations |
| **Dynamic `decimals()`** | Decimals change or are non-standard (e.g., 0/8) | Math errors in display/logic |
| **Re-entrancy in token hooks** | Calls into protocol during `transfer()` | Opens up logic reentrancy |
| **Misnamed functions** | `Transfer` instead of `transfer`, `totalSupply` returns wrong type | Broken integrations |

---

Cause

- The original ERC20 spec (EIP-20) was too vague:
  - Did not require return values
  - Did not enforce reverting on failure
  - Allowed a wide interpretation of `transfer`, `approve`, etc.
- Projects rushed to deploy tokens before stricter standards (like ERC777 or OpenZeppelin best practices) matured.

---

Where to Look

You must be extra careful in:

1. **Treasury and Token Holding Contracts**:
   - Vaults, escrows, staking pools
2. **Token Bridges and Wrappers**:
   - L2 bridges, bridge contracts, cross-chain systems
3. **AMMs, Swappers, and DEXes**:
   - Anywhere tokens are exchanged or transferred
4. **Lending Protocols**:
   - Where tokens are deposited, withdrawn, or transferred on behalf of users
5. **Any Protocol Using `transfer` / `transferFrom`**:
   - Check for assumptions: return value, revert behavior, fixed decimals

---

Why This Happens

Because `ERC20` ≠ `safe`. Unless explicitly coded for quirks:

- Protocols assume return `true`,
- Assume full amount will be transferred,
- Forget to check if `transfer()` actually succeeded.

And since these bugs **do not always revert**, the protocol may **continue execution with invalid internal state**, silently corrupting balances or enabling loss of funds.

---

Recommended Solutions

Use OpenZeppelin's `SafeERC20`

Always interact using:

```
using SafeERC20 for IERC20;

token.safeTransfer(to, amount);
token.safeTransferFrom(from, to, amount);
```

It handles:

- Missing return values
- Reverts
- Non-standard implementations

Detect and Handle Deflationary / Fee-on-Transfer Tokens

If tokens burn or charge fees on transfer:

- Measure actual balance delta before/after:

```
uint256 before = token.balanceOf(address(this));
token.safeTransferFrom(user, address(this), amount);
uint256 received = token.balanceOf(address(this)) - before;
```

Normalize or Fix Decimals

Don't hardcode `18` — always fetch using:

```
try token.decimals() returns (uint8 d) {
    // use d
} catch {
    // fallback default
}
```

Re-entrancy Safety

Assume that any token transfer **can call back into your contract**.
Use:

- Checks-Effects-Interactions pattern

- `nonReentrant` modifiers
- Avoid assuming transfer is atomic and final

Audit Dependencies for Token Assumptions

If you inherit/compose vaults, bridges, or pools, make sure downstream logic checks transfer success properly.