---

title: ERC-20 with Safety Rails description: How to help people avoid silly mistakes author: Ori Pomerantz lang: en tags: ["erc-20"] skill: beginner published: 2022-08-15

Introduction {#introduction}

One of the great things about Ethereum is that there is no central authority that can modify or undo your transactions. One of the great problems with Ethereum is that there is no central authority with the power to undo user mistakes or illicit transactions. In this article you learn about some of the common mistakes that users commit with ERC-20 tokens, as well as how to create ERC-20 contracts that help users to avoid those mistakes, or that give a central authority some power (for example to freeze accounts).

Note that while we will use theOpenZeppelin ERC-20 token contract, this article does not explain it in great details. You can find this information here.

If you want to see the complete source code:

1. Open the Remix IDE.
2. Click the clone github icon ( ).
3. Clone the github repository `https://github.com/qbzzt/20220815-erc20-safety-rails`.
4. Open **contracts > erc20-safety-rails.sol**.

Creating an ERC-20 contract {#creating-an-erc-20-contract}

Before we can add the safety rail functionality we need an ERC-20 contract. In this article we'll use the OpenZeppelin Contracts Wizard. Open it in another browser and follow these instructions:

1. Select **ERC20**.
2. Enter these settings:

| Parameter | Value | | ---------- | ---------- | | Name | SafetyRailsToken | | Symbol | SAFE | | Premint | 1000 | | Features | None | | Access Control | Ownable | | Upgradability | None |

1. Scroll up and click **Open in Remix** (for Remix) or **Download** to use a different environment. I'm going to assume you're using Remix, if you use something else just make the appropriate changes.
2. We now have a fully functional ERC-20 contract. You can expand `deps > npm` to see the imported code.
3. Compile, deploy, and play with the contract to see that it functions as an ERC-20 contract. If you need to learn how to use Remix, use this tutorial.

Common mistakes {#common-mistakes}

The mistakes {#the-mistakes}

Users sometimes send tokens to the wrong address. While we cannot read their minds to know what they meant to do, there are two error types that happen a lot and are easy to detect:

1. Sending the tokens to the contract's own address. For example,Optimism's OP token managed to accumulate over 120,000 OP tokens in less than two months. This represents a significant amount of wealth that presumably people just lost.

2. Sending the tokens to an empty address, one that doesn't correspond to anexternally owned account or a smart contract. While I don't have statistics on how often this happens,one incident could have cost 20,000,000 tokens.

Preventing transfers {#preventing-transfers}

The OpenZeppelin ERC-20 contract includes a hook, `_beforeTokenTransfer`, that is called before a token is transferred. By

default this hook does not do anything, but we can hang our own functionality on it, such as checks that revert if there's a problem.

To use the hook, add this function after the constructor:

```
solidity function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual override(ERC20)
{ super._beforeTokenTransfer(from, to, amount); }
```

Some parts of this function may be new if you aren't very familiar with Solidity:

```
solidity internal virtual
```

The `virtual` keyword means that just as we inherited functionality from `ERC20` and overrode this function, other contracts can inherit from us and override this function.

```
solidity override(ERC20)
```

We have to specify explicitly that we're overriding the ERC20 token definition of `_beforeTokenTransfer`. In general, explicit definitions are a lot better, from the security standpoint, than implicit ones - you cannot forget that you've done something if it's right in front of you. That is also the reason we need to specify which superclass's `_beforeTokenTransfer` we are overriding.

```
solidity super._beforeTokenTransfer(from, to, amount);
```

This line calls the `_beforeTokenTransfer` function of the contract or contracts from which we inherited which have it. In this case, that is only `ERC20`, `Ownable` does not have this hook. Even though currently `ERC20._beforeTokenTransfer` doesn't do anything, we call it in case functionality is added in the future (and we then decide to redeploy the contract, because contracts don't change after deployment).

Coding the requirements {#coding-the-requirements}

We want to add these requirements to the function:

- The `to` address cannot equal `address(this)`, the address of the ERC-20 contract itself.
- The `to` address cannot be empty, it has to be either:
- An externally owned accounts (EOA). We can't check if an address is an EOA directly, but we can check an address's ETH balance. EOAs almost always have a balance, even if they are no longer used - it's difficult to clear them to the last wei.
- A smart contract. Testing if an address is a smart contract is a bit harder. There is an opcode that checks the external code length, called `EXTCODESIZE`, but it is not available directly in Solidity. We have to use Yul, which is EVM assembly, for it. There are other values we could use from Solidity (`<address>.code` and `<address>.codehash`), but they cost more.

Lets go over the new code line by line:

```
solidity require(to != address(this), "Can't send tokens to the contract address");
```

This is the first requirement, check that `to` and `this(address)` are not the same thing.

```
solidity bool isToContract; assembly { isToContract := gt(extcodesize(to), 0) }
```

This is how we check if an address is a contract. We cannot receive output directly from Yul, so instead we define a variable to hold the result (`isToContract` in this case). The way Yul works is that every opcode is considered a function. So first we call `EXTCODESIZE` to get the contract size, and then use `GT` to check it is not zero (we are dealing with unsigned integers, so of course it can't be negative). We then write the result to `isToContract`.

```
solidity require(to.balance != 0 || isToContract, "Can't send tokens to an empty address");
```

And finally, we have the actual check for empty addresses.

Administrative access {#admin-access}

Sometimes it is useful to have an administrator that can undo mistakes. To reduce the potential for abuse, this administrator can be a multisig so multiple people have to agree on an action. In this article we'll have two administrative features:

1. Freezing and unfreezing accounts. This can be useful, for example, when an account might be compromised.
2. Asset cleanup.

Sometimes frauds send fraudulent tokens to the real token's contract to gain legitimacy. For example see here. The legitimate ERC-20 contract is 0x4200....0042. The scam that pretends to be it is 0x234....bbe.

It is also possible that people send legitimate ERC-20 tokens to our contract by mistake, which is another reason to want to have a way to get them out.

OpenZeppelin provides two mechanisms to enable administrative access:

- `Ownable` contracts have a single owner. Functions that have the `onlyOwner` modifier can only be called by that owner. Owners can transfer ownership to somebody else or renounce it completely. The rights of all other accounts are typically identical.
- `AccessControl` contracts have role based access control (RBAC).

For the sake of simplicity, in this article we use `Ownable`.

Freezing and thawing contracts {#freezing-and-thawing-contracts}

Freezing and thawing contracts requires several changes:

- A mapping from addresses to booleans to keep track of which addresses are frozen. All values are initially zero, which for boolean values is interpreted as false. This is what we want because by default accounts are not frozen.

```
solidity mapping(address => bool) public frozenAccounts;
```

- Events to inform anybody interested when an account is frozen or thawed. Technically speaking events are not required for these actions, but it helps off chain code to be able to listen to these events and know what is happening. It's considered good manners for a smart contract to emit them when something that miught be relevant to somebody else happens.

The events are indexed so will be possible to search for all the times an account has been frozen or thawed.

```
solidity // When accounts are frozen or unfrozen event AccountFrozen(address indexed _addr); event
AccountThawed(address indexed _addr);
```

- Functions for freezing and thawing accounts. These two functions are nearly identical, so we'll only go over the freeze function.

```
solidity function freezeAccount(address addr) public onlyOwner
```

Functions marked `public` can be called from other smart contracts or directly by a transaction.

```
solidity { require(!frozenAccounts[addr], "Account already frozen"); frozenAccounts[addr] = true; emit
AccountFrozen(addr); } // freezeAccount
```

If the account is already frozen, revert. Otherwise, freeze it and `emit` an event.

- Change `_beforeTokenTransfer` to prevent money being moved from a frozen account. Note that money can still be transferred into the frozen account.

```
solidity require(!frozenAccounts[from], "The account is frozen");
```

Asset cleanup {#asset-cleanup}

To release ERC-20 tokens held by this contract we need to call a function on the token contract to which they belong, either `transfer` or `approve`. There's no point wasting gas in this case on allowances, we might as well transfer directly.

```
solidity function cleanupERC20( address erc20, address dest ) public onlyOwner { IERC20 token = IERC20(erc20);
```

This is the syntax to create an object for a contract when we receive the address. We can do this because we have the definition for ERC20 tokens as part of the source code (see line 4), and that file includes the definition for IERC20, the interface for an OpenZeppelin ERC-20 contract.

```
solidity uint balance = token.balanceOf(address(this)); token.transfer(dest, balance); }
```

This is a cleanup function, so presumably we don't want to leave any tokens. Instead of getting the balance from the user manually, we might as well automate the process.

Conclusion {#conclusion}

This is not a perfect solution - there is no perfect solution for the "user made a mistake" problem. However, using these kind of checks can at least prevent some mistakes. The ability to freeze accounts, while dangerous, can be used to limit the damage of certain hacks by denying the hacker the stolen funds.