title: "Some tricks used by scam tokens and how to detect them" description: In this tutorial we dissect a scam token to see some of the tricks that scammers play, how they implement them, and how we can detect them. author: Ori Pomerantz tags: ["scam", "solidity", "erc-20", "javascript", "typescript"] skill: intermediate published: 2023-09-15 lang: en

In this tutorial we dissect a scam token to see some of the tricks that scammers play and how they implement them. By the end of the tutorial you will have a more comprehensive view of ERC-20 token contracts, their capabilities, and why skepticism is necessary. Then we look at the events emitted by that scam token and see how we can identify that it is not legitimate automatically.

Scam tokens - what are they, why do people do them, and how to avoid them {#scam-tokens}

One of the most common uses for Ethereum is for a group to create a tradable token, in a sense their own currency. However, anywhere there are legitimate use cases that bring value, there are also criminals who try to steal that value for themselves.

You can read more about this subject elsewhere on ethereum.org from a user perspective. This tutorial focuses on dissecting a scam token to see how it's done and how it can be detected.

How do I know wARB is a scam? {#warb-scam}

The token we dissect is wARB, which pretends to be equivalent to the legitimate ARB token.

The easiest way to know which is the legitimate token is looking at the originating organization, Arbitrum. The legitimate addresses are specified in their documentation.

Why is the source code available? {#why-source}

Normally we'd expect people who try to scam others to be secretive, and indeed many scam tokens do not have their code available (for example, this one and this one).

However, legitimate tokens usually publish their source code, so to appear legitimate scam tokens' authors' sometimes do the same. wARB is one of those tokens with source code available, which makes it easier to understand it.

While contract deployers can choose whether or not to publish the source code, they *can't* publish the wrong source code. The block explorer compiles the provided source code independently, and if doesn't get the exact same bytecode, it rejects that source code. You can read more about this on the Etherscan site.

Comparison to legitimate ERC-20 tokens {#compare-legit-erc20}

We are going to compare this token to legitimate ERC-20 tokens. If you are not familiar with how legitimate ERC-20 tokens are typically written, see this tutorial.

Constants for privileged addresses {#constants-for-privileged-addresses}

Contracts sometimes need privileged addresses. Contracts that are designed for long term use allow some privileged address to change those addresses, for example to enable the use of a new multisig contract. There are several ways to do this.

The `HOP` token contract uses the `Ownable` pattern. The privileged address is kept in storage, in a field called `_owner` (see the third file, `Ownable.sol`).

```
solidity abstract contract Ownable is Context { address private _owner; . . . }
```

The `ARB` token contract does not have a privileged address directly. However, it does not need one. It sits behind a `proxy` at address `0xb50721bcf8d664c30412cfbc6cf7a15145234ad1`. That contract has a privileged address (see the fourth file, `ERC1967Upgrade.sol`) that be used for upgrades.

```
solidity /** * @dev Stores a new address in the EIP1967 admin slot. */ function _setAdmin(address newAdmin) private {
require(newAdmin != address(0), "ERC1967: new admin is the zero address"); StorageSlot.getAddressSlot(_ADMIN_SLOT).value =
newAdmin; }
```

In contrast, the `wARB` contract has a hard coded `contract_owner`.

```
solidity contract WrappedArbitrum is Context, IERC20 { . . . address deployer = 0xB50721BCf8d664c30412Cfbc6cf7a15145234ad1;
address public contract_owner = 0xb40dE7b1beE84Ff2dc22B70a049A07A13a411A33; . . . }
```

This contract owner is not a contract that could be controlled by different accounts at different times, but an externally owned account.

This means that it is probably designed for short term use by an individual, rather than as a long term solution to control an ERC-20 that will remain valuable.

And indeed, if we look in Etherscan we see that the scammer only used this contract for only 12 hours (first transaction to last transaction) during May 19th, 2023.

The fake `_transfer` function {#the-fake-transfer-function}

It is standard to have actual transfers happen using an internal `_transfer` function.

In `wARB` this function looks almost legitimate:

```` ```solidity function _transfer(address sender, address recipient, uint256 amount) internal virtual{ require(sender != address(0), "ERC20: transfer from the zero address"); require(recipient != address(0), "ERC20: transfer to the zero address"); ````

```
    _beforeTokenTransfer(sender, recipient, amount);

    _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
    _balances[recipient] = _balances[recipient].add(amount);
    if (sender == contract_owner){
        sender = deployer;
    }
    emit Transfer(sender, recipient, amount);
}
```

```` ``` ````

The suspicious part is:

`solidity if (sender == contract_owner){ sender = deployer; } emit Transfer(sender, recipient, amount);`

If the contract owner sends tokens, why does the `Transfer` event show they come from `deployer`?

However, there is a more important issue. Who calls this `_transfer` function? It can't be called from the outside, it is marked `internal`. And the code we have doesn't include any calls to `_transfer`. Clearly, it is here as a decoy.

```` ```solidity function transfer(address recipient, uint256 amount) public virtual override returns (bool) { ````*f*(_msgSender(), recipient, amount); return true; }

```
function transferFrom(address sender, address recipient, uint256 amount) public virtual override returns (bool) {
    _f_(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer amount exceeds allowance"));
    return true;
}
```

```` ``` ````

When we look at the functions that are called to transfer tokens, `transfer` and `transferFrom`, we see that they call a completely different function, `_f_`.

The real `_f_` function {#the-real-f-function}

```` ```solidity function ````*f*(address sender, address recipient, uint256 amount) internal*mod*(sender,recipient,amount) virtual { require(sender != address(0), "ERC20: transfer from the zero address"); require(recipient != address(0), "ERC20: transfer to the zero address");

```
    _beforeTokenTransfer(sender, recipient, amount);

    _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
    _balances[recipient] = _balances[recipient].add(amount);
    if (sender == contract_owner){

        sender = deployer;
    }
    emit Transfer(sender, recipient, amount);
}
```

```` ``` ````

There are two potential red flags in this function.

- The use of the function modifier `_mod_`. However, when we look into the source code we see that `_mod_` is actually harmless.

`solidity modifier _mod_(address sender, address recipient, uint256 amount){ _; }`

- The same issue we saw in `_transfer`, which is when `contract_owner` sends tokens they appear to come from `deployer`.

The fake events function `dropNewTokens` {#the-fake-events-function-dropNewTokens}

Now we come to something that looks like an actual scam. I edited the function a bit for readability, but it's functionally equivalent.

```
solidity function dropNewTokens(address uPool, address[] memory eReceiver, uint256[] memory eAmounts) public auth()
```

This function has the `auth()` modifier, which means it can only be called by the contract owner.

```
solidity modifier auth() { require(msg.sender == contract_owner, "Not allowed to interact"); _; }
```

This restriction makes perfect sense, because we wouldn't want random accounts to distribute tokens. However, the rest of the function is suspicious.

```
solidity { for (uint256 i = 0; i < eReceiver.length; i++) { emit Transfer(uPool, eReceiver[i], eAmounts[i]); } }
```

A function to transfer from a pool account to an array of receivers an array of amounts makes perfect sense. There are many use cases in which you'll want to distribute tokens from a single source to multiple destinations, such as payroll, airdrops, etc. It is cheaper (in gas) to do in a single transaction instead of issuing multiple transactions, or even calling the ERC-20 multiple times from a different contract as part of the same transaction.

However, `dropNewTokens` doesn't do that. It emits `Transfer` events, but does not actually transfer any tokens. There is no legitimate reason to confuse offchain applications by telling them of a transfer that did not really happen.

The burning `Approve` function {#the-burning-approve-function}

ERC-20 contracts are supposed to have an `approve` function for allowances, and indeed our scam token has such a function, and it is even correct. However, because Solidity is descended from C it is case significant. "Approve" and "approve" are different strings.

Also, the functionality is not related to `approve`.

```
solidity function Approve( address[] memory holders)
```

This function is called with an array of addresses for holders of the token.

```
solidity public approver() {
```

The `approver()` modifying makes sure only `contract_owner` is allowed to call this function (see below).

```solidity for (uint256 i = 0; i < holders.length; i++) { uint256 amount = _balances[holders[i]]; _beforeTokenTransfer(holders[i], 0x0000000000000000000000000000000000000001, amount); _balances[holders[i]] = _balances[holders[i]].sub(amount, "ERC20: burn amount exceeds balance"); _balances[0x0000000000000000000000000000000000000001] = _balances[0x0000000000000000000000000000000000000001].add(amount); } }

```

For every holder address the function moves the holder's entire balance to the address `0x00...01`, effectively burning it (the actual `burn` in the standard also changes the total supply, and transfers the tokens to `0x00...00`). This means that `contract_owner` can remove the assets of any user. That doesn't seem like a feature you'd want in a governance token.

Code quality issues {#code-quality-issues}

These code quality issues don't *prove* that this code is a scam, but they make it appear suspicious. Organized companies such as Arbitrum don't usually release code this bad.

The `mount` function {#the-mount-function}

While it is not specified in the standard, generally speaking the function that creates new tokens is called `mint`.

If we look in the `wARB` constructor, we see the time mint function has been renamed to `mount` for some reason, and is called five times with a fifth of the initial supply, instead of once for the entire amount for efficiency.

```solidity constructor () public {

```
    _name = "Wrapped Arbitrum";
    _symbol = "wARB";
    _decimals = 18;
    uint256 initialSupply = 1000000000000;

    mount(deployer, initialSupply*(10**18)/5);
    mount(deployer, initialSupply*(10**18)/5);
    mount(deployer, initialSupply*(10**18)/5);
```
```

```
    mount(deployer, initialSupply*(10**18)/5);
    mount(deployer, initialSupply*(10**18)/5);
}
```

The `mount` function itself is also suspicious.

`solidity function mount(address account, uint256 amount) public { require(msg.sender == contract_owner, "ERC20: mint to the zero address");`

Looking at the `require`, we see that only the contract owner is allowed to mint. That is legitimate. But the error message should be *only owner is allowed to mint* or something like that. Instead, it is the irrelevant *ERC20: mint to the zero address*. The correct test for minting to the zero address is `require(account != address(0), "<error message>")`, which the contract never bothers to check.

`solidity _totalSupply = _totalSupply.add(amount); _balances[contract_owner] = _balances[contract_owner].add(amount); emit Transfer(address(0), account, amount); }`

There are two more suspicious facts, directly related to minting:

- There is an `account` parameter, which is presumably the account that should receive the minted amount. But the balance that increases is actually `contract_owner`'s.
- While the balance increased belongs to `contract_owner`, the event emitted shows a transfer to `account`.

Why both `auth` and `approver`? Why the `mod` that does nothing? {#why-both-autho-and-approver-why-the-mod-that-does-nothing}

This contract contains three modifiers: `_mod_`, `auth`, and `approver`.

`solidity modifier _mod_(address sender, address recipient, uint256 amount){ _; }`

`_mod_` takes three parameters and doesn't do anything with them. Why have it?

```solidity modifier auth() { require(msg.sender == contract_owner, "Not allowed to interact"); _; }

```
modifier approver() {
    require(msg.sender == contract_owner, "Not allowed to interact");
    _;
}
```

```

`auth` and `approver` make more sense, because they check that the contract was called by `contract_owner`. We'd expect certain privileged actions, such as minting, to be limited to that account. However, what is the point of having two separate functions that do *precisely the same thing*?

What can we detect automatically? {#what-can-we-detect-automatically}

We can see that `wARB` is a scam token by looking at Etherscan. However, that is a centralized solution. In theory, Etherscan could be subverted or hacked. It is better to be able to figure out independently if a token is legitimate or not.

There are some tricks we can use to identify that an ERC-20 token is suspicious (either a scam or very badly written), by looking at the events they emit.

Suspicious `Approval` events {#suspicious-approval-events}

`Approval` events should only happen with a direct request (in contrast to `Transfer` events which can happen as a result of an allowance). See the Solidity docs for a detailed explanation of this issue and why the requests need to be direct, rather than mediated by a contract.

This means that `Approval` events that approve spending from an externally owned account have to come from transactions that originate in that account, and whose destination is the ERC-20 contract. Any other kind of approval from an externally owned account is suspicious.

Here is a program that identifies this kind of event, using viem and TypeScript, a JavaScript variant with type safety. To run it:

1. Copy `.env.example` to `.env`.
2. Edit `.env` to provide the URL to an Ethereum mainnet node.
3. Run `pnpm install` to install the necessary packages.
4. Run `pnpm susApproval` to look for suspicious approvals.

Here is a line by line explanation:

```
typescript import { Address, TransactionReceipt, createPublicClient, http, parseAbiItem, } from "viem" import { mainnet }
from "viem/chains"
```

Import type definitions, functions, and the chain definition from`viem`.

```
typescript import { config } from "dotenv" config()
```

Read `.env` to get the URL.

```
typescript const client = createPublicClient({ chain: mainnet, transport: http(process.env.URL), })
```

Create a Viem client. We only need to read from the blockchain, so this client does not need a private key.

```
typescript const testedAddress = "0xb047c8032b99841713b8e3872f06cf32beb27b82" const fromBlock = 16859812n const toBlock =
16873372n
```

The address of the suspicious ERC-20 contract, and the blocks within which we'll look for events. Node providers typically limit our ability to read events because the bandwidth can get expensive. Luckily `wARB` wasn't in use for an eighteen hour period, so we can look for all the events (there were only 13 in total).

```
typescript const approvalEvents = await client.getLogs({ address: testedAddress, fromBlock, toBlock, event: parseAbiItem(
"event Approval(address indexed _owner, address indexed _spender, uint256 _value)" ), })
```

This is the way to ask Viem for event information. When we provide it with the exact event signature, including field names, it parses the event for us.

```
typescript const isContract = async (addr: Address): boolean => await client.getBytecode({ address: addr })
```

Our algorithm is only applicable to externally owned accounts. If there is any bytecode returned by`client.getBytecode` it means that this is a contract and we should just skip it.

If you haven't used TypeScript before, the function definition might look a bit weird. We don't just tell it the first (and only) parameter is called `addr`, but also that it is of type`Address`. Similarly, the `: boolean` part tells TypeScript that the return value of the function is a boolean.

```
typescript const getEventTxn = async (ev: Event): TransactionReceipt => await client.getTransactionReceipt({ hash:
ev.transactionHash })
```

This function gets the transaction receipt from an event. We need the receipt to ensure we know what was the transaction destination.

```
typescript const suspiciousApprovalEvent = async (ev : Event) : (Event | null) => {
```

This is the most important function, the one that actually decides if an event is suspicious or not. The return type,`(Event | null)`, tells TypeScript that this function can return either an `Event` or `null`. We return `null` if the event is not suspicious.

```
typescript const owner = ev.args._owner
```

Viem has the field names, so it parsed the event for us.`_owner` is the owner of the tokens to be spent.

```
typescript // Approvals by contracts are not suspicious if (await isContract(owner)) return null
```

If the owner is a contract, assume this approval is not suspicious. To check if a contract's approval is suspicious or not we'll need to trace the full execution of the transaction to see if it ever got to the owner contract, and if that contract called the ERC-20 contract directly. That is a lot more resource expensive than we'd like to do.

```
typescript const txn = await getEventTxn(ev)
```

If the approval comes from an externally owned account, get the transaction that caused it.

```
typescript // The approval is suspicious if it comes an EOA owner that isn't the transaction's `from` if (owner.toLowerCase()
!= txn.from.toLowerCase()) return ev
```

We can't just check for string equality because addresses are hexadecimal, so they contain letters. Sometimes, for example in`txn.from`, those letters are all lowercase. In other cases, such as `ev.args._owner`, the address is in mixed-case for error identification.

But if the transaction isn't from the owner, and that owner is externally owned, then we have a suspicious transaction.

```
typescript // It is also suspicious if the transaction destination isn't the ERC-20 contract we are // investigating if
(txn.to.toLowerCase() != testedAddress) return ev
```

Similarly, if the transaction's `to` address, the first contract called, isn't the ERC-20 contract under investigation then it is suspicious.

```
typescript // If there is no reason to be suspicious, return null. return null }
```

If neither condition is true then the `Approval` event is not suspicious.

```typescript const testPromises = approvalEvents.map((ev) => suspiciousApprovalEvent(ev)) const testResults = (await Promise.all(testPromises)).filter((x) => x != null)

console.log(testResults) ```

An `async` function returns a `Promise` object. With the common syntax, `await x()`, we wait for that `Promise` to be fulfilled before we continue processing. This is simple to program and follow, but it is also inefficient. While we are waiting for the `Promise` for a specific event to be fulfilled we can already get working on the next event.

Here we use `map` to create an array of `Promise` objects. Then we use `Promise.all` to wait for all of those promises to the resolved. We then `filter` those results to remove the non-suspicious events.

### Suspicious `Transfer` events {#suspicious-transfer-events}

Another possible way to identify scam tokens is to see if they have any suspicious transfers. For example, transfers from accounts that don't have that many tokens. You can see how to implement this test, but `wARB` doesn't have this issue.

## Conclusion {#conclusion}

Automated detection of ERC-20 scams suffers from false negatives, because a scam can use a perfectly normal ERC-20 token contract that just doesn't represent anything real. So you should always attempt to *get the token address from a trusted source*.

Automated detection can help in certain cases, such as DeFi pieces, where there are many tokens and they need to be handled automatically. But as always caveat emptor, do your own research, and encourage your users to do likewise.
```