De.Fi

SECURITY BIBLE

How to Outplay 90% DeFi Users
That Don’t Know the Basics of DeFi Security

www.de.fi

Content

What is this book about?

The Security Bible is brought to you by the DeFi Pioneers and the Authors of the World's First book about DeFi and the Amazon Bestseller - the "Wall Street Era is Over".

Despite hundreds of security audit providers and whitehat hackers trying to do their best at preventing hacks in the crypto space, industry participants become victims of scams and exploits every day. It's not always the case that projects you consider to invest in or use for your routine DeFi operations have been analyzed for security by top audit providers and passed all crucial checks. Thus, doing at least some basic due diligence before interacting with any protocol is a must.

On one hand, DeFi scams and exploits go increasingly sophisticated. And even professional smart contract auditors can have a hard time figuring out what was a backdoor of a specific attack even after it had already happened. So, even full-on manual auditing doesn't guarantee prevention of all possible risks.

On the other hand, we can talk about some major patterns and signs helping to detect if a DeFi project is risky. Knowing them will allow you to filter out a large portion of malicious actors aiming to steal your funds. And that's the goal of this book.

We are going to show you what are frequent backdoors by contract and protocol type, how you can do on-chain analysis with easily accessible tools such as Etherscan, and extract plenty of useful security information even without reading the actual smart contract code. But we will not bypass diving into smart contract functionality as well.

What is our motivation?

We've been in DeFi since 2020, and so far we have:

- Built Crypto's First Antivirus - the De.Fi Scanner

- Built Crypto's Most Advanced Revoke Tool - the Shield
- Built World's Largest Database of Crypto Scams: REKT Database
- Built Crypto's First Database of Audits: Audit Database
- Prevented multiple scams by forcing projects to fix their smart contracts' issues, with the most notable cases being: Defrost Finance, Alpha Homora and PancakeBunny
- Performed 100+ fully trustless smart contract audits

General risk markers

Depending on project category and on contract type, there is a specific set of risk markers you should pay attention to in order to avoid rug pulls.

But first, let's look at some universal security checks applicable to any protocol and contract type. There are general red flags that can help you detect projects that are not what they appear.

Team

The project team's positioning, behavior and development history can help you understand if the project is open, community-oriented and likely to react quickly to emergency cases such as exploits or internal Dapp malfunctions.

The main factor to check here is if the team of the project you are interested in is public. There are a ton of famous and successful projects built by anonymous developers. But, for newer projects that have not proved their real intentions with time yet, anonymous team members are always a **red flag**. They are harder to come after in case of a rug pull, whereas public teams are fully responsible for business results in

accordance with legislation of countries they operate in. Public teams in law enforcement not only for assuring users' funds safety, but also for not following their reward promises and roadmap milestones.

It's also worth dedicating time into investigating connections of the project's team members to old projects that appeared to be scams or just unsuccessful businesses. **When doing your research:**

- check if addresses of contract deployers and admin roles are listed in databases of blacklisted addresses?
- examine if devs that made commits to the project's Github repositories are not mentioned in community discussions of the past years with negative connotation?

It also might be highly informative to review development history by answering:

- Is the git repository of the project open? Community and independent auditors should be able to analyze the code managing user funds.
- How old is the repo? Created a few days before launch - **red flag**.
- Is the protocol a fork? What protocol is it a fork from? If it's just a low-effort, rushed copy of a major DEX or solely a simple token with nothing built around it - definitely a **red flag**.
- How many devs are commiting to the repo and with what frequency? When was the first and the last commit?
- Is there any documentation for the protocol? Can smart contracts be easily found on the project's website, its docs, or in the README of its repository? Documentation should be detailed, covering all the protocol's facets - from user flow to explanation of smart contract functionality and dev guides for API integrations.
- When was the domain of the project's website registered?
- What is the general UX/UI feel? Scam projects usually have a poor design, a 1-page UI with a few buttons. Avoid making little to zero effort.

Social profile analysis is another step in estimating the project's team. Projects that care about their community publish frequent development updates, have multiple active communication channels, provide feedbacks on user claims and give detailed answers. Avoid projects that provide blurry information or just ignore user worries.

For certain rug pull types, analysis of team and development history is the only way to suspect the risk in time and stay away from investing in a hyped but shady project. First of all, we are talking about abandoning, which is among the most frequent ways to exit-scam.

External audits

Availability of external audits is always a plus to the project's reputation. The more eyes look at the code the better, especially when we are talking blockchain where deployed code cannot be modified, and protocol breaches and rug pulls are an everyday thing. Pay special attention to high-risk findings, make sure they have been fixed by the team.

Multiple audits present? Even better. Each auditor has its own set of metrics to assess security of DeFi projects. Some auditors only perform automated checks, others do manual reviews or combine automated and manual techniques. Some are only focused on checking smart contracts vulnerabilities, others have extended scope and analyze non-code project characteristics such as on-chain information, governance, development history etc., along with the technical part. The ability to see different security estimation results and compare them enables getting the fullest picture about the project's advantages and disadvantages from the security perspective. By the way, you can find audits done for a project in De.Fi's audit database. All audits from all providers are gathered in one place to simplify your due diligence and provide you with great comparability of security auditing results.

But, you should understand that auditing is not always requested by projects for a real in-depth security check, and audit providers are not always interested in making sure projects they analyze are: a) decentralized, transparent, user-oriented; b) not vulnerable to exploits.

Nowadays, there are hundreds of smart contract auditors. The biggest part of them provide low-quality services. Their security knowledge is poor, they don't provide clear explanations on discovered risks, don't collaborate with developers to help them improve analyzed smart contracts, or just flood audit results with unverified warnings revealed with automatic tools such as Slither and Mythril.

Moreover, potentially risky contracts might not get into auditing scope the project and the auditor provider negotiated about. A vulnerability or a backdoor can remain overlooked until an attack happens or an independent community auditor gets their hand on it.

Nevertheless, the fact that the project at least bothered paying a significant sum to auditors is a positive sign. This filters out a big pile of smaller scammy projects.

Usage of coin mixers

Make sure coin mixers, such as Tornado Cash and Typhoon Cash, were not used to fund the protocol you are analyzing. This is a huge **red flag**. Mixers are often utilized by scam project teams in order to remain untraceable after deployment of their malicious contracts and stealing user money.

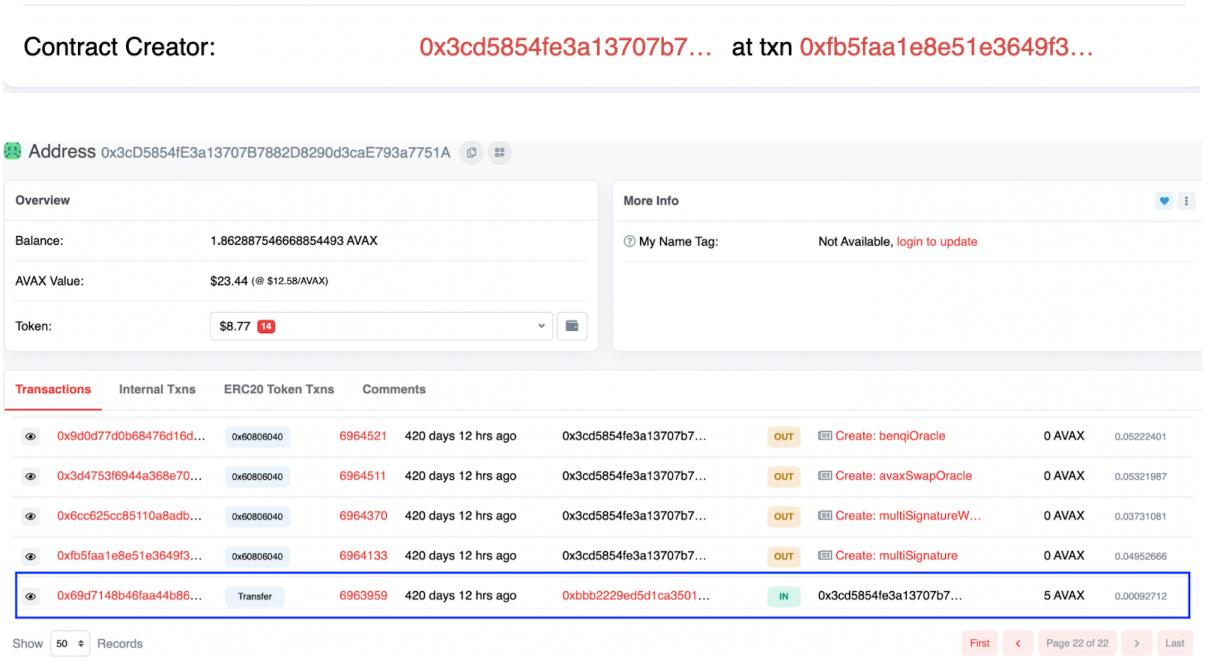
All mixers function in a similar way. They allow users to transfer funds between two addresses without any onchain proof that they are linked. In this way, transfer transactions get anonymized.

Mixers are smart contracts working as pools where users deposit their funds. The latter get mixed together and then withdrawn to new addresses.

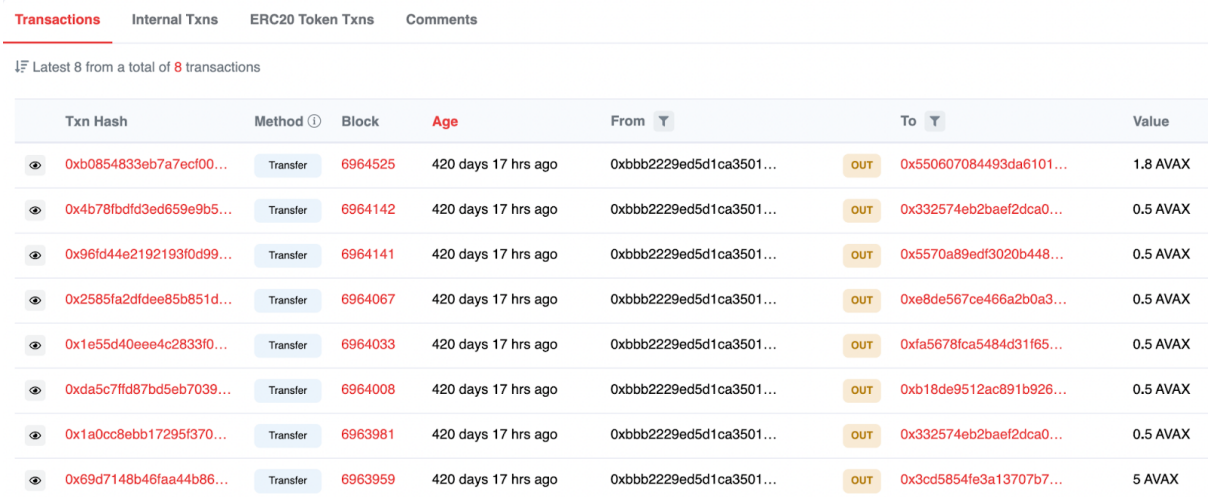
How to detect the risk?

Here we need to find out how the contract deployer received funds to pay gas for deployment to blockchain. This can be easily done with a tool available to all - Etherscan.

After checking who is the deployer of the analyzed contract, open their address and look for a transaction where they received native currency to cover gas for further transactions.



We can see that AVAX came from an EOA - 0xbbb2229ed5d1ca3501F8C4ef31741b6800e12d3b. Now let's check its transactions.

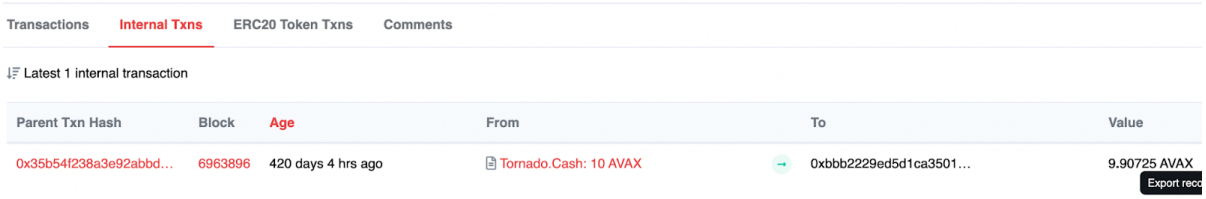


Transactions | Internal Txns | ERC20 Token Txns | Comments

Latest 8 from a total of 8 transactions

Txn Hash	Method	Block	Age	From		To	Value
0xb0854833eb7a7ecf00…	Transfer	6964525	420 days 17 hrs ago	0xbbb2229ed5d1ca3501…	OUT	0x550607084493da6101…	1.8 AVAX
0x4b78fbdfd3ed659e9b5…	Transfer	6964142	420 days 17 hrs ago	0xbbb2229ed5d1ca3501…	OUT	0x332574eb2baef2dca0…	0.5 AVAX
0x96fd44e2192193f0d99…	Transfer	6964141	420 days 17 hrs ago	0xbbb2229ed5d1ca3501…	OUT	0x5570a89edf3020b448…	0.5 AVAX
0x2585fa2dfdee85b851d…	Transfer	6964067	420 days 17 hrs ago	0xbbb2229ed5d1ca3501…	OUT	0xe8de567ce466a2b0a3…	0.5 AVAX
0x1e55d40eee4c2833f0…	Transfer	6964033	420 days 17 hrs ago	0xbbb2229ed5d1ca3501…	OUT	0xfa5678fca5484d31f65…	0.5 AVAX
0xda5c7ffd87bd5eb7039…	Transfer	6964008	420 days 17 hrs ago	0xbbb2229ed5d1ca3501…	OUT	0xb18de9512ac891b926…	0.5 AVAX
0x1a0cc8ebb17295f370…	Transfer	6963981	420 days 17 hrs ago	0xbbb2229ed5d1ca3501…	OUT	0x332574eb2baef2dca0…	0.5 AVAX
0x69d7148b46faa44b86…	Transfer	6963959	420 days 17 hrs ago	0xbbb2229ed5d1ca3501…	OUT	0x3cd5854fe3a13707b7…	5 AVAX

Here, we don't see any receipts of the native coin. But, what if we look into the Internal Transactions section, and here it is:



Transactions | Internal Txns | ERC20 Token Txns | Comments

Latest 1 internal transaction

Parent Txn Hash	Block	Age	From		To	Value
0x35b54f238a3e92abbd…	6963896	420 days 4 hrs ago	Tornado.Cash: 10 AVAX	→	0xbbb2229ed5d1ca3501…	9.90725 AVAX

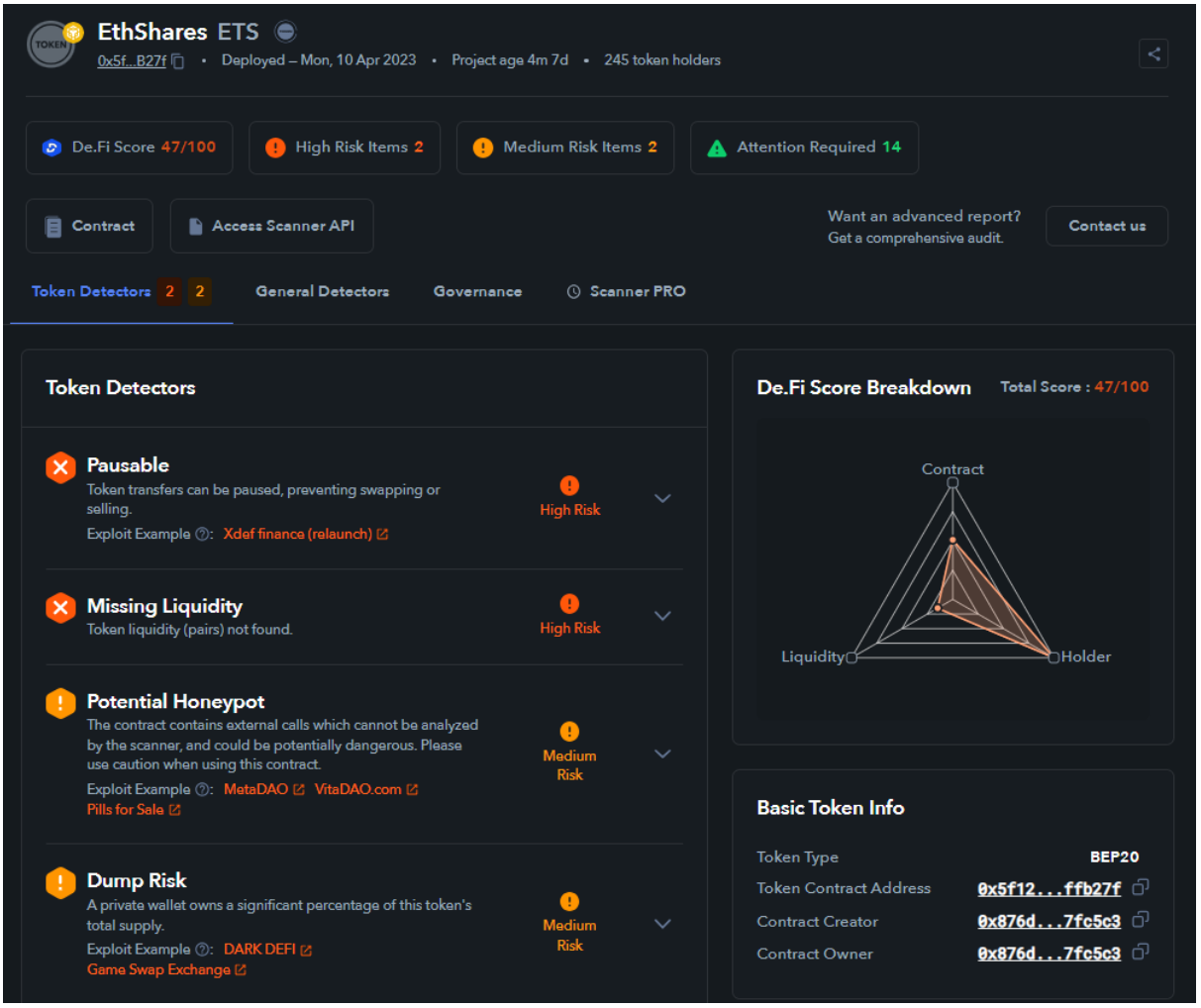
Export reco

Unverified smart contract code

Verified smart contract code is a sign of the project's transparency. Unverified code means that source code of analyzed contracts has not been published by its deployer, and users can only see smart contract bytecode. In this case, it's impossible to have a clue what the contracts really do. Even if the project contacts are published on Github, there is no proof that they match to the actually deployed code. Any malicious functionality can be hidden there. If you dare to interact with an unverified contract, you probably are ready to be scammed.

There are exceptions though. Some popular protocols may keep some part of their contracts unverified, or they might verify contracts on one chain but keep them unverified on another one.

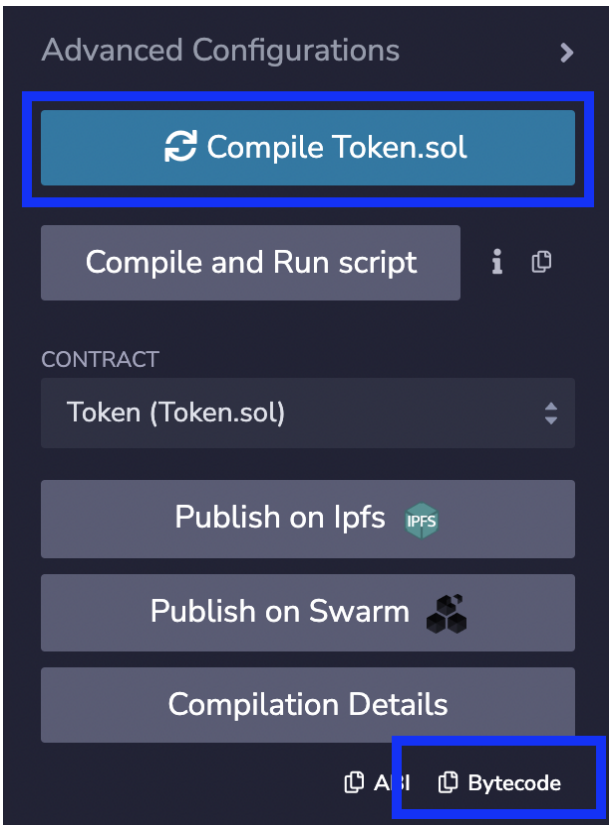
If you still consider using an unverified contract in light of other factors about the project being positive, you can insure yourself with scanning it using De.Fi Scanner.



Firstly, the tool checks a few sources for the deployed code. Even if a contract is not verified on a blockchain explorer (Etherscan, Bscscan etc.), it might be verified on Tenderly. Secondly, you'll still be able to see results of the transactions you intending to sign. Don't approve any smart contract interactions if you are not confident about the consequences. Take your time in analyzing who will be the real receiver of a requested token transfer.

If you feel you can go more advanced, you can search for a Github repository of the project with the contract you are about to interact with. You'll need to compile the found source code to bytecode. One of the ways

to do it fast is by using Remix IDE. Just create a .sol file with the source code taken from Github, compile it and copy the generated bytecode.



https://remix.ethereum.org

Now you can compare the obtained bytecode to the one you see on the blockchain explorer for the unverified contract. If they are not identical, devs of the contract deployed some other code to the blockchain than they published in their repository.

By the way, if you've found the source code of a contract which is not verified on the blockchain explorer yet, you can perform its verification yourself. Just submit the source code in a few easy steps.

Code upgradability

As smart contract code deployed to the blockchain cannot be modified, some dev teams decide to use upgradability patterns called proxies. How do they work? Two contracts are deployed. One acts like an interface and info storage, another one contains logic that must be executed. The one that users interact with is a proxy. When developers want to change logic, they just deploy a new logic contract called implementation, which gets bound to the proxy. The proxy address remains the same so users won't notice anything.

While this brings flexibility for projects utilizing this pattern, users get under a significant risk as functionality of a contract holding their funds or having access to their wallets through approvals can be changed in a malicious way in order to do a rugpull.

Governance

Another part of your DeFi due diligence should analyze who runs the protocol's crucial functionality. We are talking changing parameters such as reward rates, protocol fees, adding strategy addresses to vaults that eventually hold user funds, setting emergency states and calling critical smart contract functionalities that influence movement of crypto assets between addresses (token minting, token migrations from one contract to another etc.) and lock/unlock user actions (pausing selected functions, blacklisting user addresses etc.).

Basically, what we've described **is governance**. And it's better be decentralized.

Decentralized governance is one of the central pillars of decentralized finance. This is an on-chain or off-chain mechanism enabling users holding the governance token of a protocol to influence decision making regarding its potential changes such as functionality updates and

parameter adjustments. The more tokens a user holds, the bigger voting power they have.

On-chain governance

It's a good sign if a project choses to implement a certain decentralized governance mechanism, especially on-chain - a DAO. On-chain governance is implemented directly on blockchain based on rules defined in governance contracts. Thus, it's a more transparent and fair way of user voting as results cannot be hidden or maliciously modified.

The industry standard for governance contracts is the Governor contract originally developed by Compound. It contains all the main features required for a function in a target contract to be called. First, a proposal is created. Then users have to vote for it within a defined time period. Only if the proposal is supported by a sufficient number of votes, it will be passed for execution with a timelock delay.

Off-chain governance

With off-chain governance, voting happens on dedicated websites like Snapshot. There is no direct connection between an accepted proposal and its execution in a corresponding contract. The future of the proposal depends on the dev team's actions. This means that the executor of administrative contract functions is a regular wallet - a so-called externally owned account (EOA), or a Multisignature wallet - a wallet with multiple owners that have to authorize a transaction to make it pass.

Centralized governance

It's a **red flag** when a project's contracts are run just by one address, a so-called externally owned account (EOA), which is a regular wallet. This means that that wallet can call privileged functions of the contracts anytime. The risk degree depends on how critical these functions are, what can be the worst scenario. For example, if the wallet can only set a protocol fee within reasonable constant limits, there is no risk here. But, if it can

replace critical addresses the contract interacts with, such as price oracles and vault strategies, user assets get under a direct danger.

Risks associated with calling privileged functions by an EOA can be mitigated if a Timelock delay is applied. It won't protect users from possible issues though. But as execution of risky functions gets delayed, users get time to react if they track the contract calls. Of course, the delay should exceed 24h. Otherise, it's pretty useless.

Sometimes you'll see contracts are controlled by multisignature wallets. They are often seen as a more safe way to manage contracts, but not in terms of rug pulls.

When a Multisig is set as the owner/admin of a contact, the risk degree of privileged functions depends on who are owners of the multisig wallet. We can say the risks are low only when owners of the multisig are verified addresses of the public team members. Moreover, it's important to check how many transaction confirmations are needed. What is the threshold? If confirmation of 1 or 2 addresses is enough to make a transaction pass, there is no risk mitigation. A few addresses can belong to one same person.

Let's think about an example: in a case when the threshold is set to 2 and the 2 addresses are anonymous, these addresses can belong to one person. Thus, risk-wise, it's the same as the contract would be managed by an EOA.

But multisigs possessing the governance role are definitely better than EOAs in terms of protection from external exploits. It's recommended to set admin roles to multisigs cause it will be harder for hackers to compromise private keys.

How to detect governance type and possible risks?

You might find all you need in the project's docs. When projects have DAO, they usually mention this in their documentation and provide guidance on

how to be a part of it and use your DAO rights: governance token is well presented, voting procedure is well explained.

If the governance section is skipped and not even described in a few words in the docs, most probably governance of the project analyzed is centralized. Thus, to get an idea of how the project is run, a set of on-chain checks are necessary.

Some risks can only be detected and correctly interpreted taking into account both static information (smart contract code) and dynamic information (parameters, possibility of their change, smart contract access control roles and ownership). But searching for on-chain information and picking up necessary pieces of smart contract code can be a significant challenge for non-tech users. This problem can be solved with De.Fi's scanner. It's designed to accelerate your due diligence and protect your crypto assets.

And you can get a deeper insight into reasons why some contracts get flagged as high, medium or high risk with advanced view

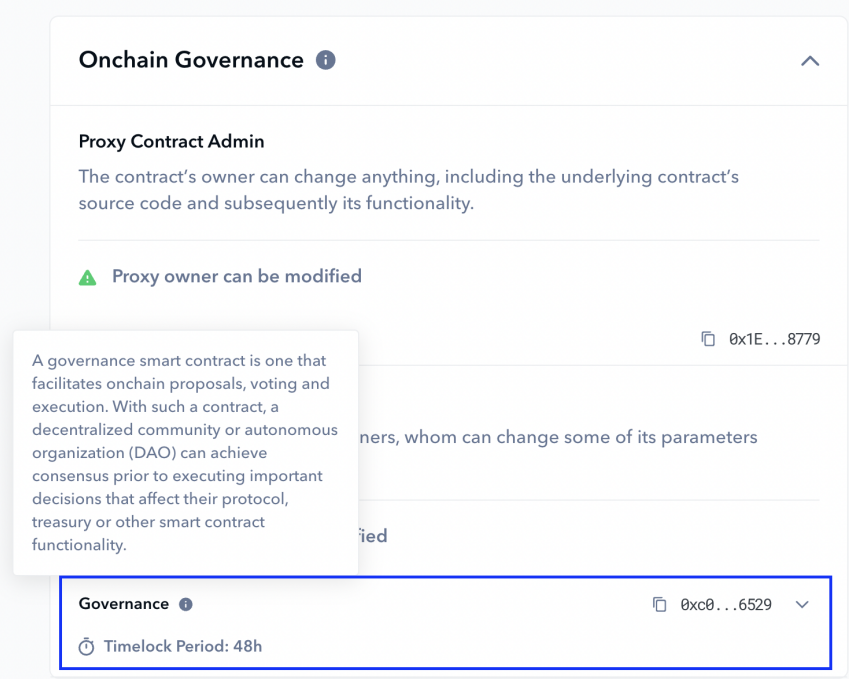
With the scanner advanced section, you can see:

- Exact location of the problematic code. Even if you are not a developer or auditor, you might at least get the general logic of a backdoor better and educate yourself to boost your investment due diligence and understand public analytics of professionals like auditors and blockchain devs better.
- Governance - the caller of the function detected as a risky one. This is one of the main context defining factors in smart contract security. We'll have a closer look at what is governance and why it is important to analyze in the dedicated section "Centralized Governance" and in analyzing examples for different smart contract risks.

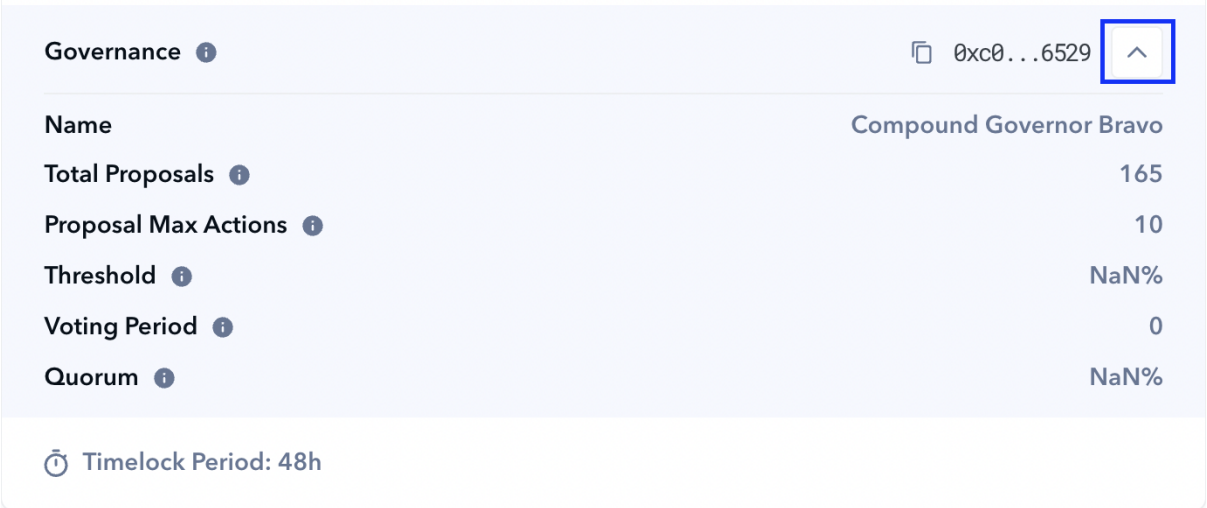
In a dedicated Governance section you can see the summary of what is the governance type, if it is modifiable and what are privileged functions that are controlled by the detected governance.

As an example let's take scan results for the contract with the address 0x316f9708bB98af7dA9c68C1C3b5e79039cD336E3. This is one of Compound's contracts - ConfiguratorProxy.

The De.Fi Scanner detected that the analyzed contract is run by on-chain governance with a 48h timelock delay. This means that any function with access control limited to the governance address can only be called when a sufficient number of governance token holders (COMP) vote for it.



You can also get details for the detected governance contract:



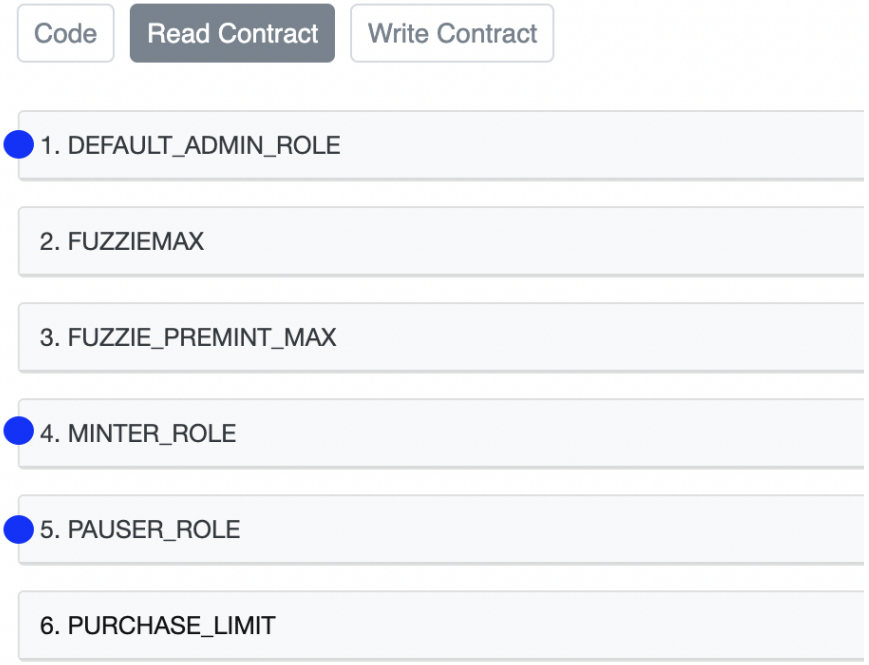
As each situation depends on the context and no automated solution can puzzle out the whole picture, sometimes it's important to have a deeper look into the results and analyze what happens under the hood. Let's see how you can do that with Etherscan (or similar solutions you prefer):

1. Find addresses of the main contracts of the protocol and open them in a blockchain explorer;
2. For each contract, try to check who is its owner. If ownership is set through the Ownable contract of the Openzeppelin library, it will be easy. Just open the Read section of the contract and click on the Owner variable:

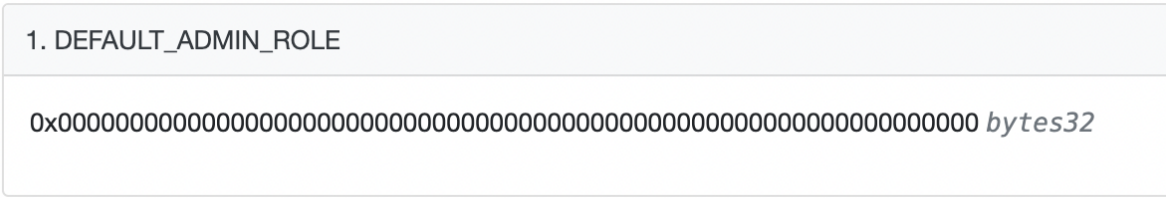
Another popular way to set access control over contract functionality is using AccessControl - another Openzeppelin's contract. It is used to set the

Admin role and add any other custom role. It's a bit more tricky in terms of reading addresses of set roles as the library's contract can be implemented with additional extensions or without. It will require an additional effort from you, but it's extremely useful for you as for a DeFier to get your hands dirty in Etherscan. So let's take a look at the next example. For demonstration, we are taking a random contract.

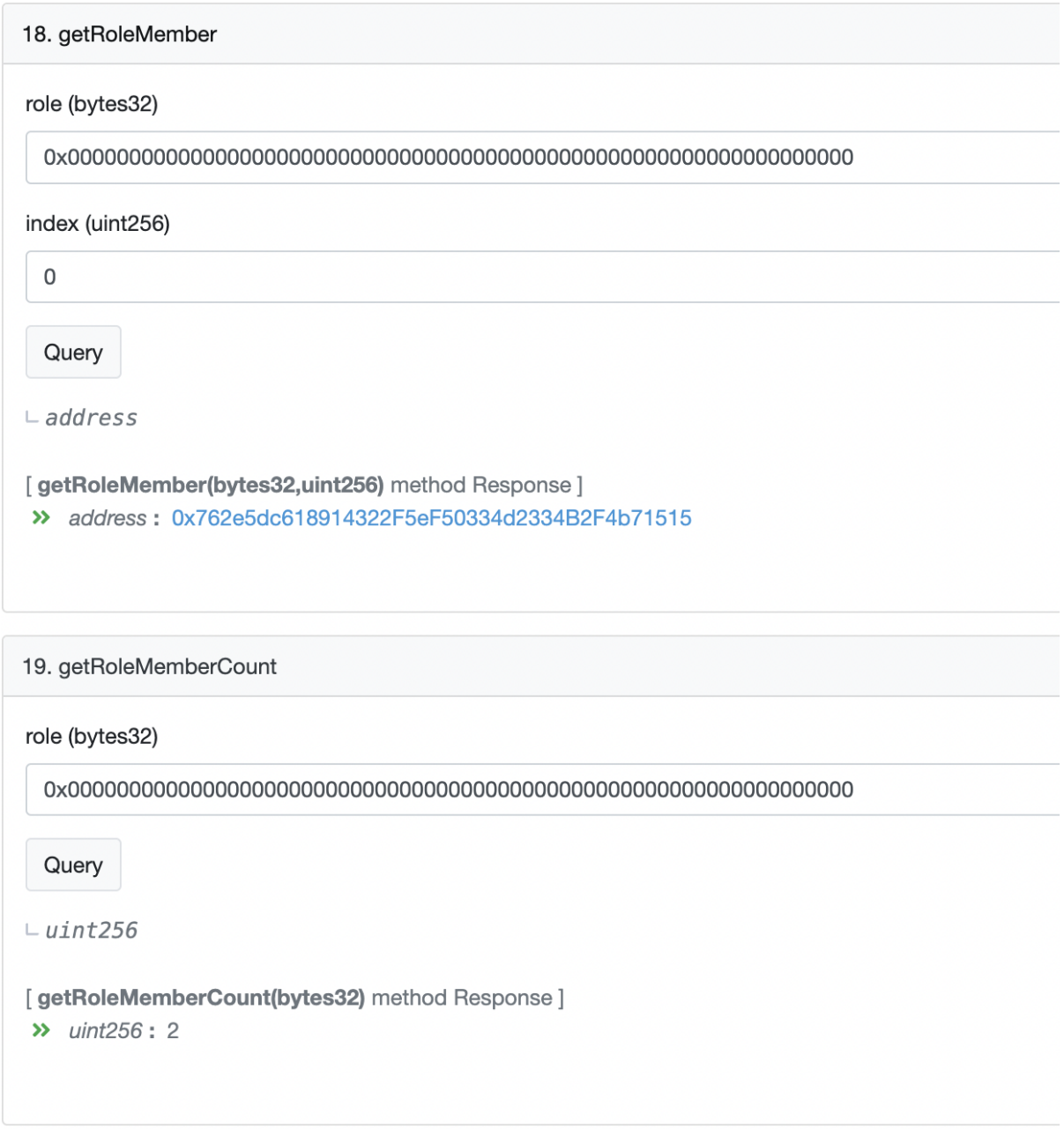
Again, open the Read section of the contract you are analyzing. If you see DEFAULT_ADMIN_ROLE and other set role variables, most probably the standard AccessControl construct was implemented.



You will need to copy bytes representation of the role you check. For the default admin role, it looks like this:



Copy this line of bites. And if AccessControl extensions are used, you will see getRoleMemberCount and getRoleMember functions. Use them as follows:



First, we get that there are 2 addresses set to the Admin role. And after calling getRoleMember, we see the first admin's address. Set index to 1 and you'll see the second's admin address.

Try yourself. You'll see that both admin addresses are EOAs. This means that they are eligible to call some set of privileged functions, including creating new roles:

```
107
108     function addToWhitelist(address[] memory add) public {
109       require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "Must have DEFAULT_ADMIN_ROLE to add to whitelist");
110         for (uint i = 0; i < add.length; i++) {
111             whitelistAddresses[add[i]] = true;
112         }
113     }
114
115     function removeFromWhitelist(address[] memory remove) public {
116       require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "Must have DEFAULT_ADMIN_ROLE remove from whitelist");
117         for (uint i = 0; i < remove.length; i++) {
118             whitelistAddresses[remove[i]] = false;
119         }
120     }
121
```

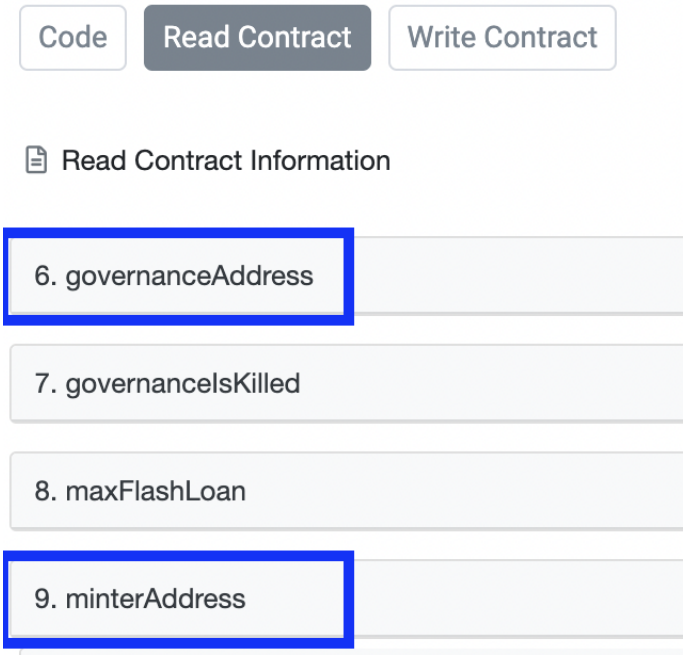
File 1 of 26: AccessControl.sol

```
158      *
159      * Requirements:
160      *
161      * - the caller must have ``role``'s admin role.
162      */
163     function grantRole(bytes32 role, address account) public virtual override onlyRole(getRoleAdmin(role)) {
164         _grantRole(role, account);
165     }
166
```

https://etherscan.io/address/0xcec4d283aa944fbce8abf6aba8d0f7af37f570f4#code

If there are no Read functions for getting the exact address/es of existing roles, either stick to an assumption that the deployer address is Admin. Additionally, you can check transaction history for grantRole() calls. If you found such calls, review details of the transactions. What addresses have been granted what roles?

Oftentimes, privileged roles are defined in a custom way: by just creating variables for needed roles and adding requirements on any functions that only those roles can call them. Here is an example:



Once you click on governanceAddress, you'll see that it is an EOA, and it can call a set of privileged functions such as defining the minter address:

```
12 contract Oxd is ERC20, Governable, ERC20Permit, ERC20FlashMint {
13     address public minterAddress;
14
15     constructor() ERC20("OXD", "OXD") ERC20Permit("OXD") {}
16
17     modifier onlyMinter() {
18         require(
19             msg.sender == minterAddress,
20             "Ownable: caller is not the minter"
21         );
22         _;
23     }
24
25     function setMinter(address _minterAddress) public onlyGovernance {
26         minterAddress = _minterAddress;
27     }
28
29     function mint(address to, uint256 amount) public onlyMinter {
30         _mint(to, amount);
31     }
32 }
```

https://ftmscan.com/address/0xc5a9848b9d145965d821aaec8fa32aaee026492d#readContract

We hope you understand the idea. Just play with Etherscan and you'll get a whole lotta useful information.

3. Examine what exactly are privileged functions of each role? What consequences may they have? Again, let's take an example. Imagine you are analyzing a vault contract. Assets that you deposit in it are sent to a strategy that generates yield. What if the vault has a setStartegy() function with the onlyAdmin modifier and the admin is EOA? Anytime, this EOA can replace the strategy address with any wanted wallet or malicious contract. And all funds deposited by users will be directed to that new "strategy" and lost forever. Even if the team of the project having this vault is public, and the devs won't have malicious intentions, the private key of the admin EOA can be compromised. So setting an EOA to be a vault admin is not a good option either way. Regarding external attacks, it at least should be a multisignature wallet. And the best option is, of course, a governance contract as it won't allow anybody to have a centralized control over users' funds.

Phishing

Phishing is when a bad actor pretends to be a well-known, trusted project in order to get sensitive information of victims, such as their wallets' private keys, or force them to do critical actions like interacting with scammers' smart contracts or malicious versions of famous applications.

The most frequent ways scammers use include:

- Sending phishing emails leading to fake websites, where users will share their seed phrases, private keys or call transactions on malicious smart contracts, which can lead to approving their balances to these scam contracts, directly sending transactions via them, or signing malicious messages. We'll talk about these actions in detail right in this section;
- Spreading harmful links in social media;
- Luring victims to malicious websites through fake Google ads hoping that they will be inattentive and won't notice the difference between the legit domain and a fake one;
- Tricking victims into downloading a fake crypto wallet presenting it as a required version update of a real wallet such as Trustwallet or Metamask.

Now let's take a closer look at token approvals and signing messages with your wallet as these are action phishing attackers always try to induce DeFi users to.

Token approvals

Understanding of this topic is crucial for security of your DeFi activity. And it is relevant **not only in the context of phishing** on original protocols but **for all DeFi scam types**.

In order to interact with any Dapp, you need to provide the latter with a right to spend underlying tokens from your balance. In other words, you have to approve a contract of that Dapp to be a spender of a particular ERC20 token you own.

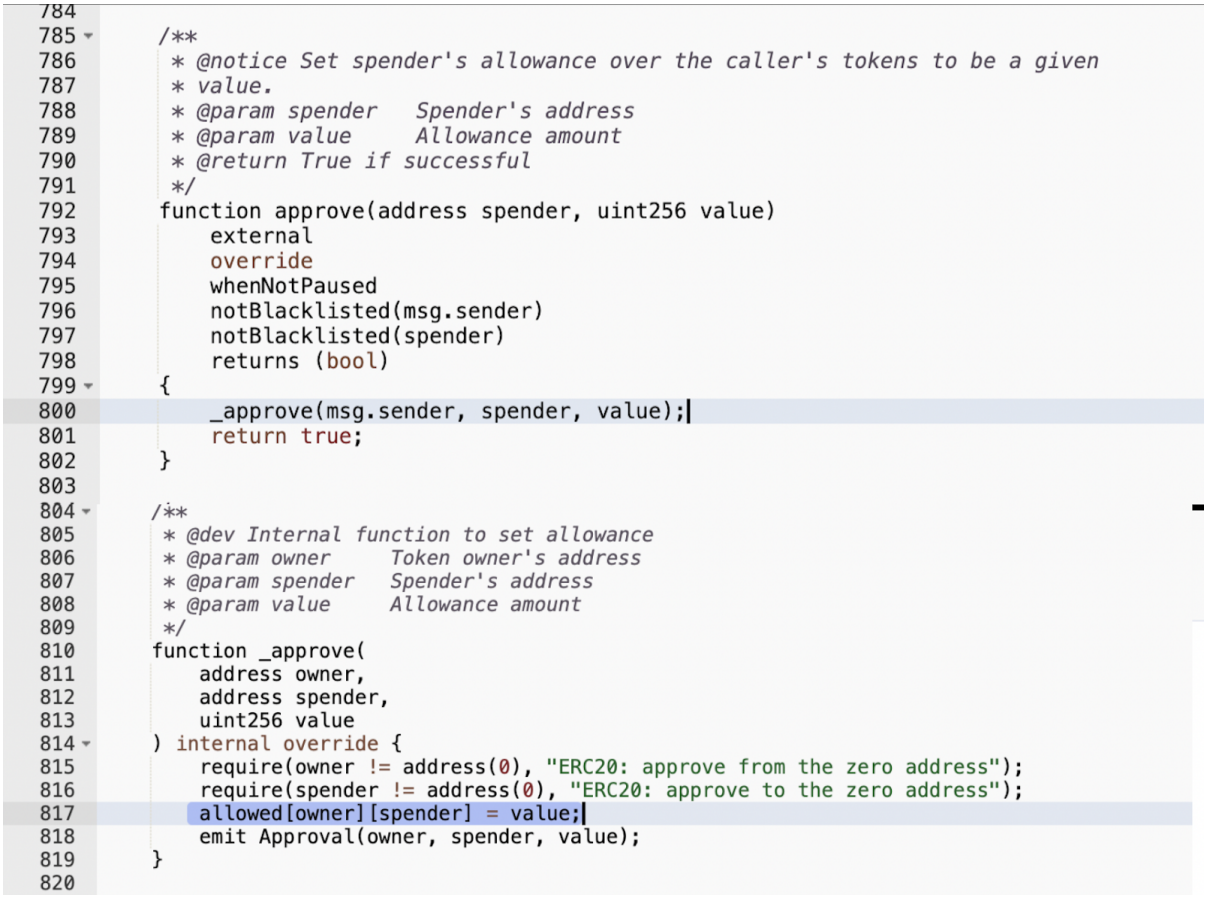
How does it work technically? Let's say you are about to deposit USDC into a vault. A typical deposit function should involve transferring the staking token from your balance to a strategy bound to the vault so that yield can be generated on the deposited amount. This is done through calling the transferFrom() function - a standard function each token based on the ERC20 standard has.

```
function transferFrom(address from, address to, uint256 amount) public virtual override returns (bool) {
    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

But only contracts approved by you can call a token transfer from your balance. Thus, prior to actually depositing, swapping, lending, placing buy/sell orders or calling any other functionality that should send your tokens to a particular target, you have to call the approve() function on the contract of the token.

In our example, you have to call approve() on the USDC contact:

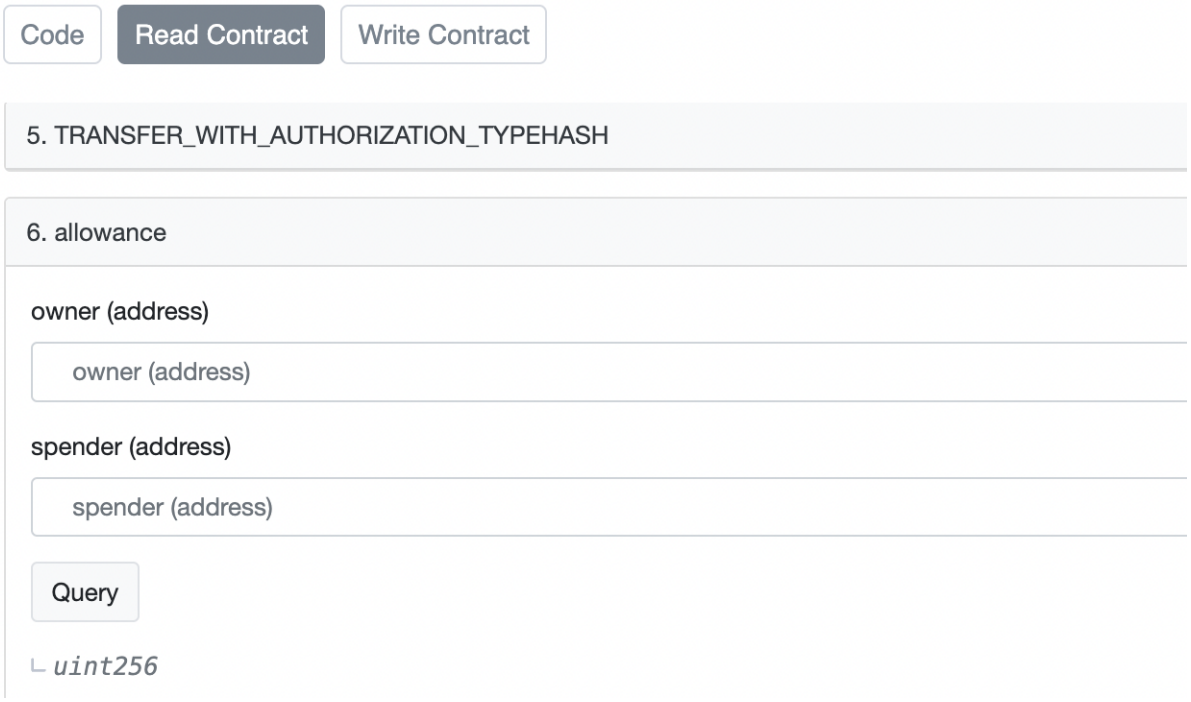


```
784
785     /**
786      * @notice Set spender's allowance over the caller's tokens to be a given
787      * value.
788      * @param spender   Spender's address
789      * @param value     Allowance amount
790      * @return True if successful
791      */
792     function approve(address spender, uint256 value)
793         external
794         override
795         whenNotPaused
796         notBlacklisted(msg.sender)
797         notBlacklisted(spender)
798         returns (bool)
799     {
800         _approve(msg.sender, spender, value);
801         return true;
802     }
803
804     /**
805      * @dev Internal function to set allowance
806      * @param owner     Token owner's address
807      * @param spender   Spender's address
808      * @param value     Allowance amount
809      */
810     function _approve(
811         address owner,
812         address spender,
813         uint256 value
814     ) internal override {
815         require(owner != address(0), "ERC20: approve from the zero address");
816         require(spender != address(0), "ERC20: approve to the zero address");
817         allowed[owner][spender] = value;
818         emit Approval(owner, spender, value);
819     }
820
```

https://etherscan.io/address/0xa2327a938febf5fec13bacfb16ae10ecbc4cbdcf#code

In the image above, the owner parameter would be your address. “Spender” is the Dapp contract that should call trasferFrom() later on when you will call deposit().

To manually check if you approved a contract to spend your tokens, you can open the Read section of the ERC20 token contract and input your address and the Dapp contract address into the view function “allowance”. It will show you the approved token amount:



https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48#readProxyContract

Now you see that contacts that have your approvals for specific tokens literally have control over your balance of those tokens.

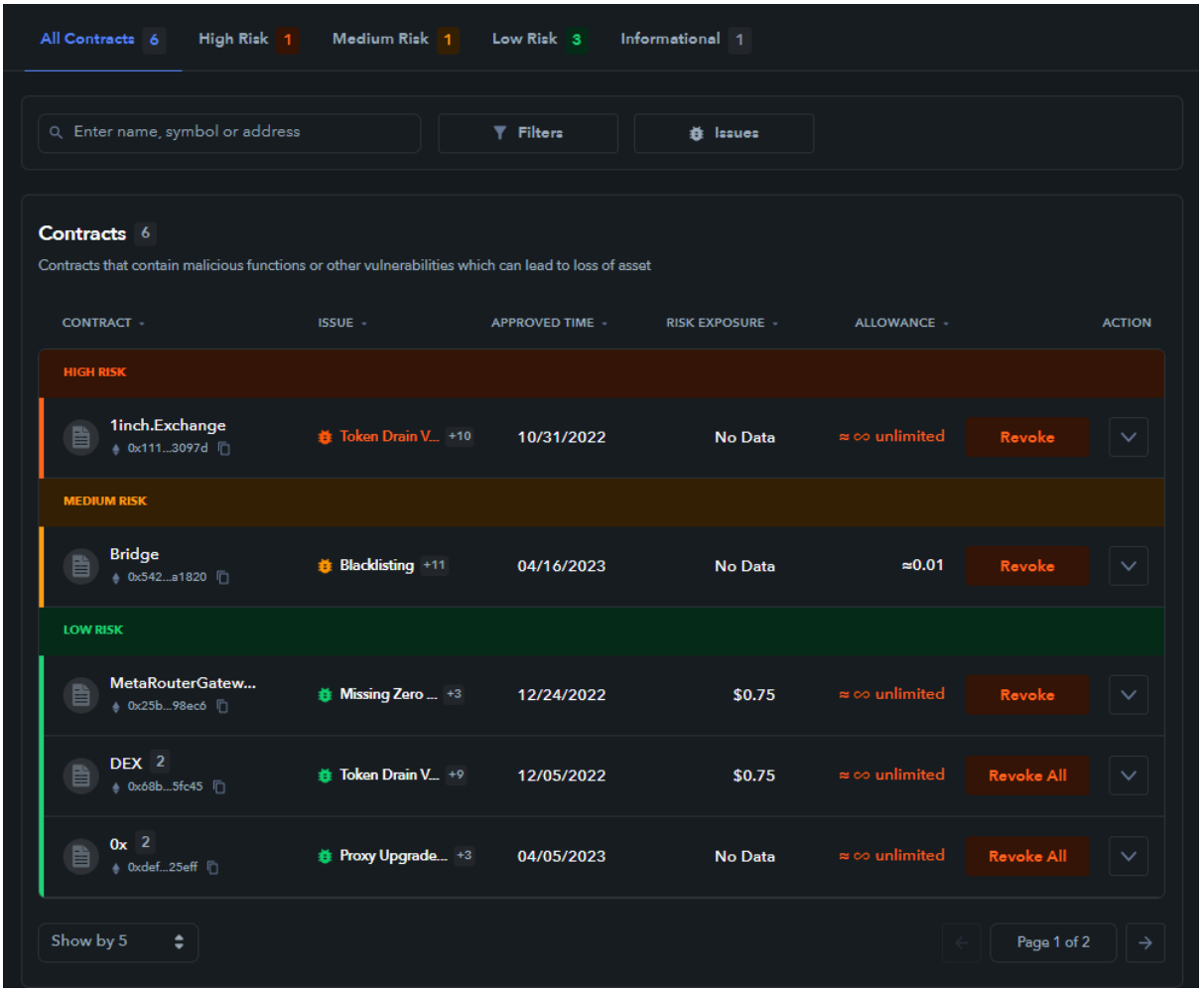
Moreover, most Dapps prefer to request unlimited token approvals from users so that they don't have to call approve() repeatedly. On the one hand, it is convenient and gas efficient. On the other hand, users get exposed to higher risks.

Imagine you approved a malicious contact to be a spender of your tokens? Yes, you definitely will lose all the balance of the token you approved.

What if you keep an unlimited token approval for a contact that has a vulnerability and can get hacked? Yes, it's possible hackers will find a way to transfer all tokens from users that have previously approved their balances to the breached contract.

That being said, we strongly recommend you keep an eye on approved contacts you have collected.

Luckily, you don't need to iterate through all contacts you might have ever interacted with and ask somebody to analyze them for possible risks. The De.Fi Shield is the best solution for checking if some of your approved contacts have high risk issues.



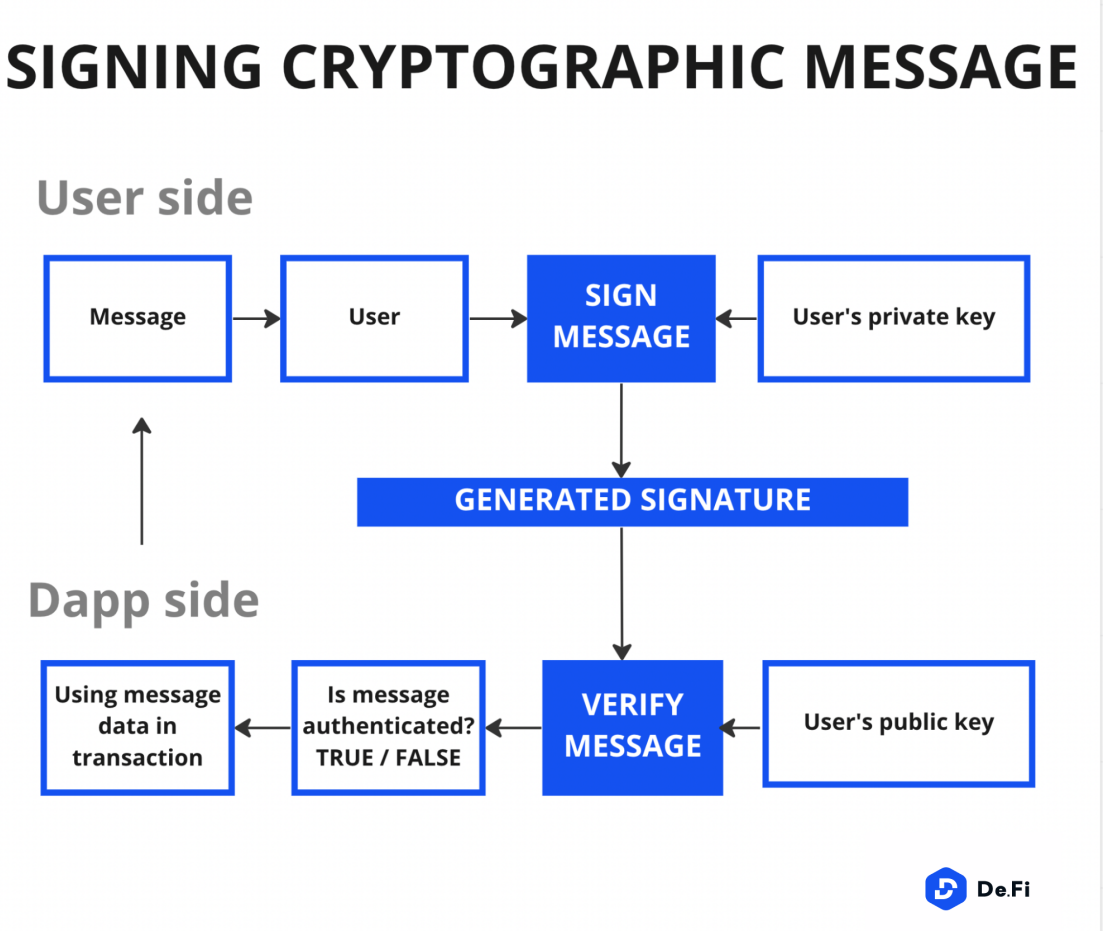
Shield scans all your approved contracts for over 80 security issues including the ones referring to high and critical risk such as possibility of a token balance drain through ERC20 approval.

Another important thing to point out is that if you've used a contact and are not going to interact with it on a regular basis, better remove all related token approvals. Unexpectedly, this contact can get exploited. You never know. Why risking your balance extra time?

Signing messages

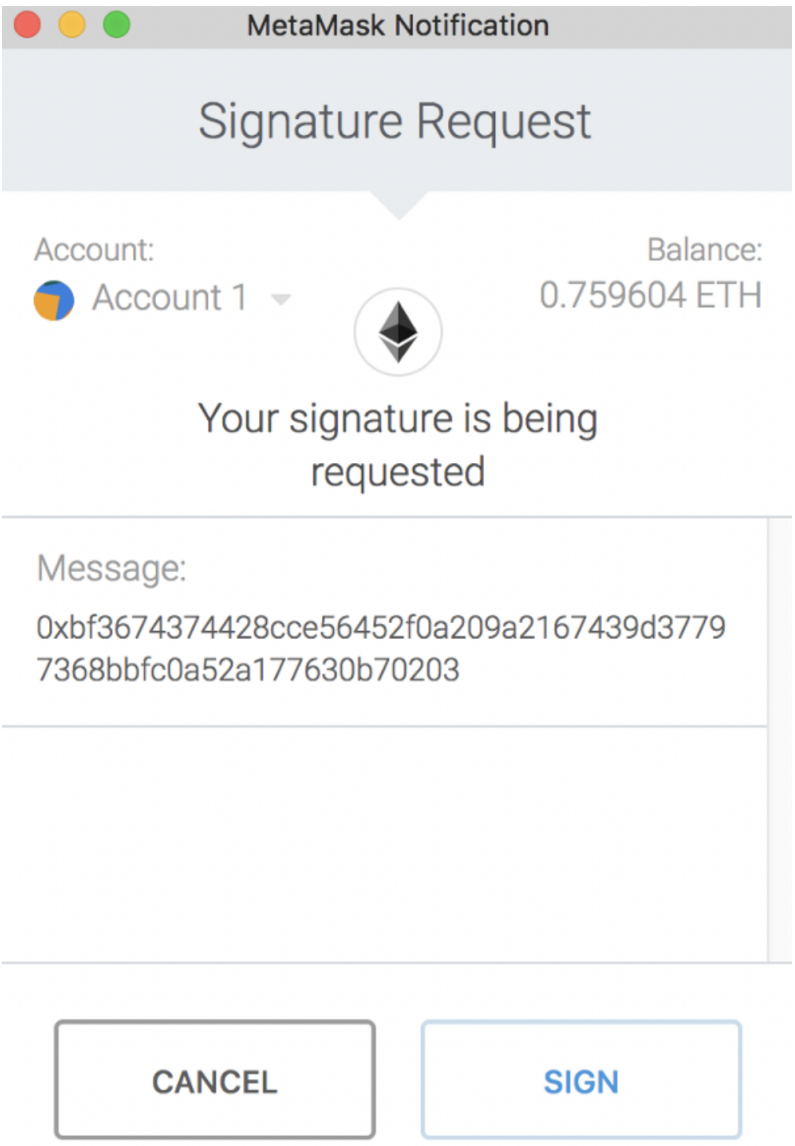
If you're into DeFi, you most probably came across processes requiring you to sign a message as a part of verification into Dapps, DAO voting, placing sell and buy orders on DEXes and NFvT marketplaces etc. Cryptographic signatures are a perfect solution when transactions must be authenticated as they don’t oblige users to spend gas.

What happens under the hood is that users receive message data, and with their private keys, they confirm that they received that data and authenticate it. In other words, signatures allow Dapps to link data of messages to user public keys with guarantee the messages are signed with private keys of those users.



Is important to Note that private keys can not be derived from signatures, so the signing mechanism is safe for users to use, but, of course, if users understand what they are signing. If they don't, there is a big chance that scammers will take advantage of their inattentiveness.

Some signature methods can be unreadable as they deliver data hashed: users just get a set of symbols like this:



But even when Dapps use signature methods providing the most comprehensible message info, such as EIP-712, where data in the message is shown in a human-readable way, many users still might not understand what parameter means what and therefore can get tricked by scammers.

Opensea has been using the ERP-712 message method since the protocol's migration to Wyvern V2 and till the currently valid SeaPort contract. Why are we mentioning Opensea now? Because one of the biggest phishing scams happened to Opensea and in general to the DeFi space was performed exactly through prompting victims to sign cryptographic messages. Malicious messages. As a result, NFTs valued at around $1.7M USD had been stolen from 17 users.

Let's analyze how it happened so that you understand the importance of signing messages and avoid similar scam patterns.

The fact that a message data is really linked to the needed public key is decoded in the target contract function. On Opensea, this decoding happens when an order gets executed. When a seller creates a sell order, they have to make 2 steps:

- Approving all items of the corresponding NFT collection to a Opensea contract;
- Signing a message with order listing data such as price, token ID, collection address, seller address and a few other parameters.

For buy orders, the flow is similar. The buyer has to approve WETH to Opensea so that the marketplace can transfer the price amount from the buyer's balance to the seller's wallet once the order match happens.

So yes, Opensea orders get created offchain. Otherwise it would be overly expensive to create, change and cancel them. The order execution itself, of course, must be paid by the transaction finalizer - a user accepting a buy or a sell order. But until the order is executed, it can be canceled absolutely for free.

The signature generated from the seller proves that their public key is linked to the order and the order parameters are accepted from them. And this is where the scammers phishing on Opesea sneaked in.

The scammers exploited the fact that orders can be created outside of the Opensea platform and can still be read by the original Opensea exchange

contract. Using phishing emails, they navigated users to requests to sign an order message, in which the malicious price parameter was palmed off. Not realizing what is hapening, the victims had confirmed with their private keys that they were ready to sell their NFTs for **0 ETH**. And the orders with the 0 ETH price were successfully processed by the Opensea exchange contract, passing all the underlying assets from the deceived users directly to the scammer. As simple as that.

Now you know possible consequences of signing messages you don't understand and don’t doubcheck.

The same fake message pattern can be applied by scammers for any limit orders on DEXEs. They can trick users to sell their ERC20s for little to zero price. Beware of what you are signing whatever Dapp you are using.

How not to get rekt from phishing?

- The first few links you see as when browsing for a DeFi protocol are not necessarily legit ones. In fact, those might be exactly directing you to scams. Take an extra minute to check the validity of the domain you are switching to. One of the ways to do that is going to the official Twitter profile of the project and taking the website link there.
- If this is a protocol you are using on a regular basis, save its website in your browser’s bookmarks.
- Using anti-phishing codes that allows users to easily distinguish emails from real DApps and their fakes. This is a unique set of characters that must come with every email a user gets from a project in order to confirm the sender is the expected Dapp.
- Be attentive to email addresses that send you links and suspicious warings. They might look similar to right contacts with only slight differences in spelling or inclusion of specific characters. Never click on links received in such a way.

- Never give your Secret Recovery Phrase to any website or someone directly asking you to in Discord or Telegram and pretending to be project admins.
- Never sign any cryptographic messages outside of original protocol websites.
- Never sign any cryptographic messages you don't understand content of.
- If you are realizing you've just signed a malicious message, you **still can save** your assets by removing approvals for contracts of the project the scammers are phishing on. Just go through each contact referring to the attacked project that you approved and cancel the approvals using De.Fi's Shield.

Fake airdrops

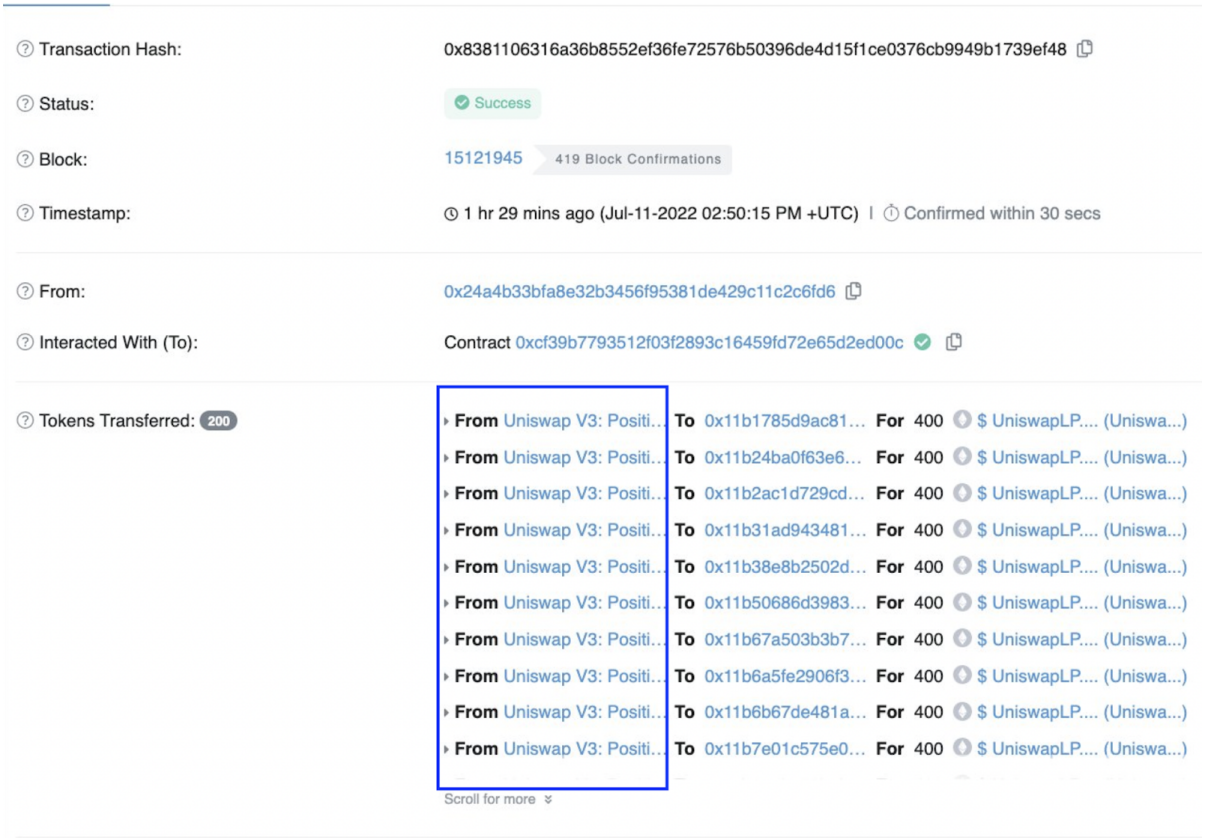
Oftentimes, phishing goes hand in hand with fake airdrops, which are the next to talk about.

If you suddenly detect some new token in your wallet, don't be jumping for joy and don't rush to swap them for some other token as "the sudden gift" can be a fake airdrop. This is a scam technique when fake tokens imitating legit, trusted tokens are sent to users of the real projects.

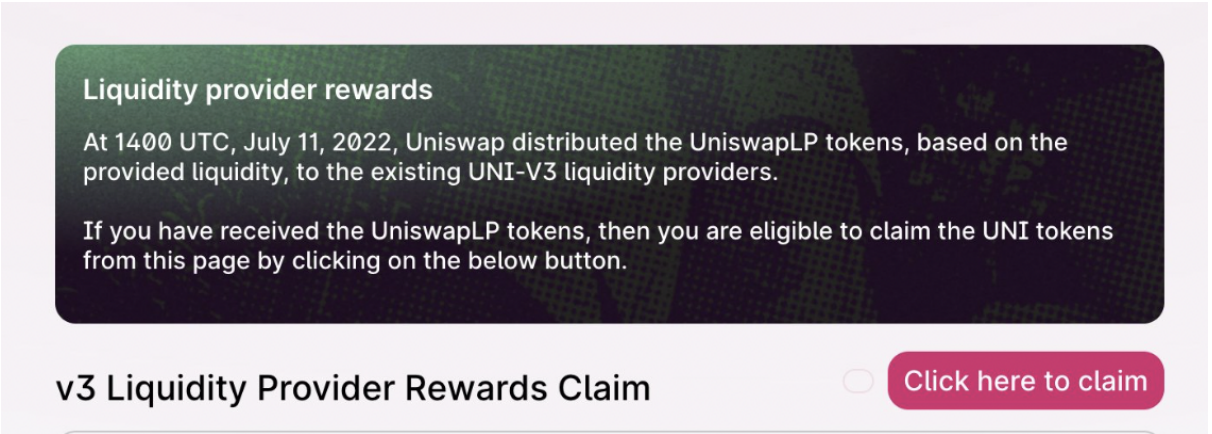
When doing fake airdrops, scammers want you to start interacting with their malicious contracts, which eventually will allow them to take control over your funds.

For you to understand how dangerous it can be, let's recall a fake UNI airdrop that happened in July 2022. Scammers sent fake Uniswap LP tokens to 73 399 Uniswap users and promised they can be redeemed to the actual UNI token.

When sending malicious LPs, the scammer also utilized the event pollution technique: event data of the attack transaction was set in a way that block explorers identified the "from" parameter as the real "Uniswap V3: Positions NFT" contract.



Of course, a large number of users got serious about it and clicked on the “UniswapLP” token received. The token name had a link redirecting to a domain called "uniswaplp.com" imitating the real Uniswap website. When opening it, users could see a claim button:



Upon clicking it, users were asked to confirm an approval transaction that gave the scammer access to all their real Uniswap V3 LPs that are required to withdraw assets that had been provided as liquidity to trading pairs.

The attacker was able to drain the real Uniswap LPs from wallets of all users that had signed the harmful transaction, and withdrew all the underlying liquidity. The stolen funds had been laundered with the privacy mixer Tornado Cash.

Similar schemes were used for fake airdrops performed on Bored Ape Yacht Club NFTs, $RUNE, $OP and tens of other famous tokens.

How not to get rekt?

- Don't interact with anything unexpected that you see on your wallet if it has not been announced in official channels of the project this airdrop claims to be a part of.
- If you've found an announcement, make sure it was posted by real admins and in at least a few official channels of the project. Take your time to contact the project's tech support if you have doubts.
- Keep in mind that scammers may perform fake airdrops simultaneously or closely in time with official airdrops of legit projects.
- Be attentive to domain names. Airdrops that try to redirect you to unofficial domains might be phishing attacks.

Classic token rugpulls

Now let's dive into token security details. In order to perform even some basic security analysis, it's important to understand what tokens are from a technical point of view.

Tokens are smart contract representations of some assets, shares, money or services that can be interacted with and exchanged through a set of standard functions. These functions are embodied in the ERC20 standard that was developed by Ethereum developers for security and development ease.

The standard ERC20 functionality includes:

- transferring tokens between accounts;
- checking token balance of an address;
- getting the token total supply currently available on the chain;
- approving tokens to be spent by other addresses other than their holders' addresses.

By following the standard, projects make sure their tokens will be fully compatible with any DeFi infrastructure: from farms and lending protocols to gameFi and Metaverses.

With libraries like OpenZeppelin contracts, ERC standards can be just copied and combined with custom functionality given the nature of the project.

Problems come when standard token functionality gets maliciously modified or extended with backdoors for rugpulls.

In this section we'll review ways of stealing funds from users that are called the classic rug pulls in DeFi. They are easy to understand and, unfortunately, to implement. Thus, we see them happening nearly every day.

Minting

Minting functionality can be risky when a large amount of tokens can be minted to an EOA, a multisig with anonymous owners and low confirmation

threshold or an unverified contract with a purpose of dumping the token on an exchange.

Privileged infinite mint

Malicious minting can be done in a few ways. Creating a privileged mint function without any limitations on the amount and destination of minted tokens is one of them.

This rug pull has a simple technique: 1. Token deploy → 2. Liquidity pair creation → 3. Token mint → 4. Token dump.

Creating some basic ERC20 token is not a big deal. You don't even need deep knowledge of Solidity to do it. There are many token constructors available for free, such as Openzeppelin Wizard, where you can get your token code just in a few minutes after selecting functionality you would need, including a privileged mint.

In terms of development, nothing else is really necessary to perform a mint rug pull. Most rugpulled tokens didn't even have any infrastructure built around them. In practice, promises on the bright token's future are enough to generate hype in most cases.

For users to start buying a newly created token, scammers then create a trading pair or few on a DEX. It's super easy and fast as it doesn't require passing any KYC process. The only thing needed is to provide liquidity for the pair to enable swaps, where the pair creator has to add some amount of the scamcoin and an appropriate amount of a valuable token like USDT and USDC given the start price of that scamcoin.

As users start buying the scamcoin, its price rises. Here is when minting a large amount of tokens comes into play. Upon receiving them to their address, scammers swap them to a valuable token. The token sell off leaves all other token holders with a token that neither has any value, nor can be sold.

Another way for scammers to get a large token balance is **embedding mint** into some unexpected protocol functionality. For example, fraudulent devs can add minting into the transfer() function of the token contract or into a Masterchef's deposit(). As a result, each time users do simple transfers or deposits, a certain amount of the project's token is minted to a wallet, controlled by the scammers.

One more trick scammers use to take control of a large portion of the token supply is preminting it to their wallet when deploying the ERC20 token contract. And the other rug pull steps remain the same: creating a liquidity pair and selling those preminted tokens when the token's price increases.

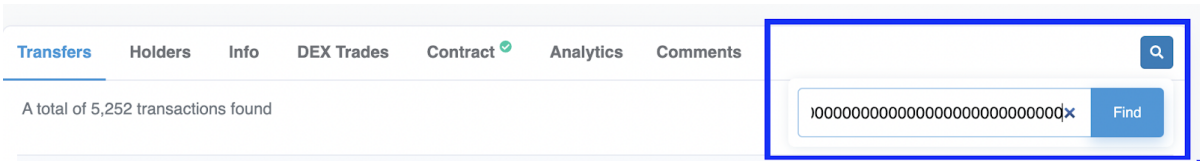
Additionally, you should keep in mind that projects can do hidden minting. Basically, what technically is minting in ERC20 tokens? Increasing balance of an address + corresponding increase of the total supply variable + emitting transfer event with "from" set to the zero address. So hidden minting happens without updating the total supply, bypassing some formal limitations if present.

Minting can be hidden in an unexpected custom function. Moreover, this backdoor can be added through modifying some standard libraries like Safemath (a lib for performing basic math operations with protection from overflows or underflows). This makes it even harder to detect. Especially for users with average to low technical knowledge.

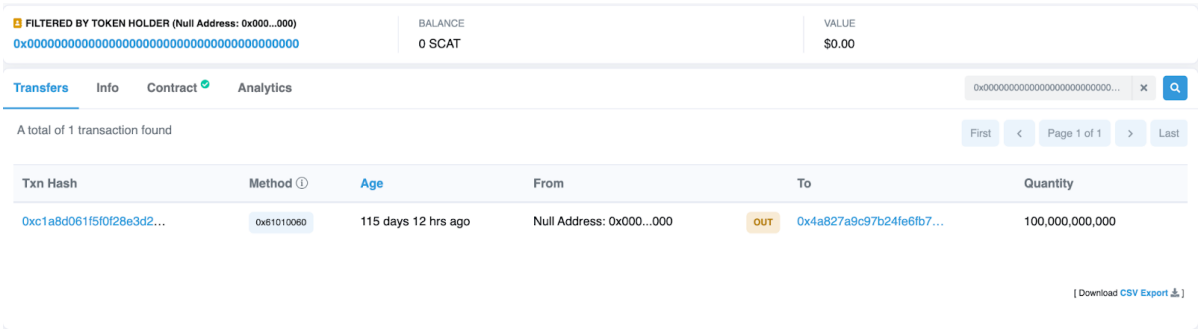
How not to get rekt?

To reveal if there was a large **premint** of a token analyzed:

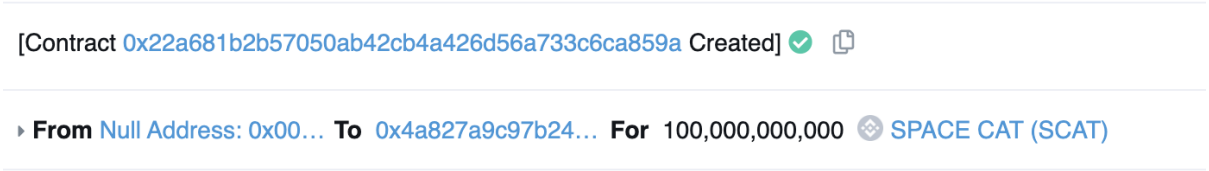
1. Go to the Transfers section of the token on a blockchain explorer;
2. Filter transfer transactions with zero address. All transfers coming FROM 0x0000000000000000000000000000000000000000 are mints:



We've taken the rugged $SCAT as an example, and here is the result:



3. Click on the transaction hash to see details. In the example, we see that the premint of 100B $SCAT happened right with deployment of the token contract and the amount was directed to the deployer address:



In order to check for possibility of privileged minting:

1. Try to find mint() in the token contract.
2. Check if only a defined role can call it. Most frequently, the allowed minters are either Owner, or admin, or Minter or Governance, or it can be a list of minter addresses.

   If it's a standard mint function, it will look like this:

```
208 }
209
210 function mint(address account, uint256 amount) public {
211     require(minters[msg.sender], "!minter");
212     _mint(account, amount);
213 }
214
```

In the example code above, there is a "require" statement that the caller must belong to a list of allowed minters. And the key word "public" means this function can be called from anyone outside of the contract with the applied "require" limitations. Often you'll see this requirement to be

implemented through the “onlyOwner”/”onlyMinter”/”onlyAdmin”/etc. modifiers.

Lets see what happens inside the internal _mint() function:

```
73    function _mint(address account, uint amount) internal {
74        require(account != address(0), "ERC20: mint to the zero address");
75
76        _totalSupply = _totalSupply.add(amount);
77        _balances[account] = _balances[account].add(amount);
78        emit Transfer(address(0), account, amount);
79    }
```

Total supply of the token gets increased by the input amount, so does the receiver’s balance. And a Transfer event is created.

But you won’t always see such a clear picture. Minting can be implemented in a totally custom way so that it’s presence is not so obvious for regular users.

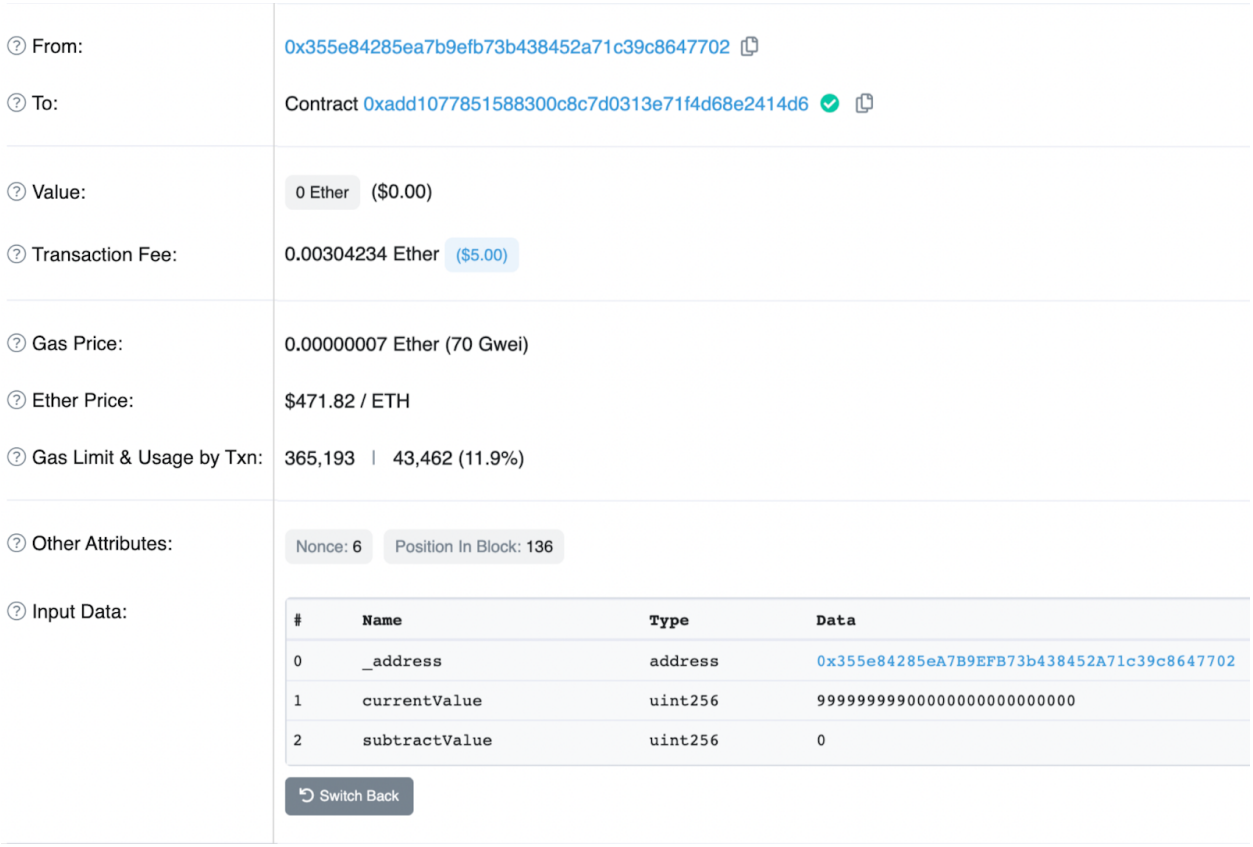
Look at how creative it can get:

```
82  function balanceOf(address _address) public view returns (uint256 balance) {
83    return balances[_address];
84  }
85  function burn (address _address, uint256 currentValue, uint256 subtractValue) external onlyOwner {
86    require(_address != address(0));
87    balances[_address] = currentValue.sub(subtractValue);
88  }
```

Yes, what you see is actually a mint! A **hidden** mint. In this “burn” function, the contract owner can literally set the token balance of any address to any value. As no event of transferring the token from zero address gets emitted and the total supply value is not updated, when the scammer calls it, it’s very unlikely somebody will notice a mint has just happened.

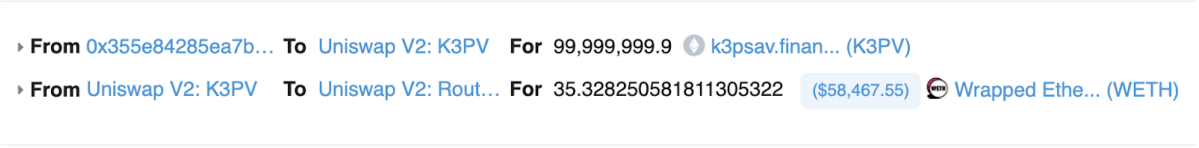
Such hidden mint won’t even appear if you filter transfer transactions in a blockchain explorer by 0x0 address. Neither you will be able to see this mint with other tools for onchain analysis like Bloxy.

By the way, the example of this creative mint function is the rug pulled $K3PV. And this is how the scammer actually used the hidden mint/”burn” backdoor:



https://etherscan.io/tx/0x4da1b53ea24cd99cf51af1668ba069d12bb6d7a618c2d083b6b7e65306d6b602

The scammer assigned 99.9M tokens to their own balance when calling this “burn”. In the next transaction, by the rug pull classic, the malicious receiver of the minted amount just swapped it on Uniswap for ETH.



https://etherscan.io/tx/0x7c78ac94ee7feb5474d75d22c5950d663ad7bd5ad393c972f40a3e9386244fb7

3. But before judging if a found mint is a risk factor, especially if it is a standard, not customly written ERC20 mint, you should check what type of address exactly is set as the minter. EOA? Unverified contract? Multisig of anonymous devs? - red flags if there are no limitations on mint(). What is common for all three named caller types? - They are centralized. Nobody can predict and influence their actions.

4. Speaking of limits, does the amount that can be input into the mint() function have limits? What are possible destinations of tokens that are about to be minted? Can the “to” parameter be set to any address?

Liquidity removal

Another classic way to rug pull - removing liquidity after a token’s price soared and leaving users with the token that’s worth nothing and can not be sold anywhere.

How not to get rekt?

Likely, performing a simple on-chain analysis is what is needed to detect a potential liquidity removal rug pull. You need to make sure:

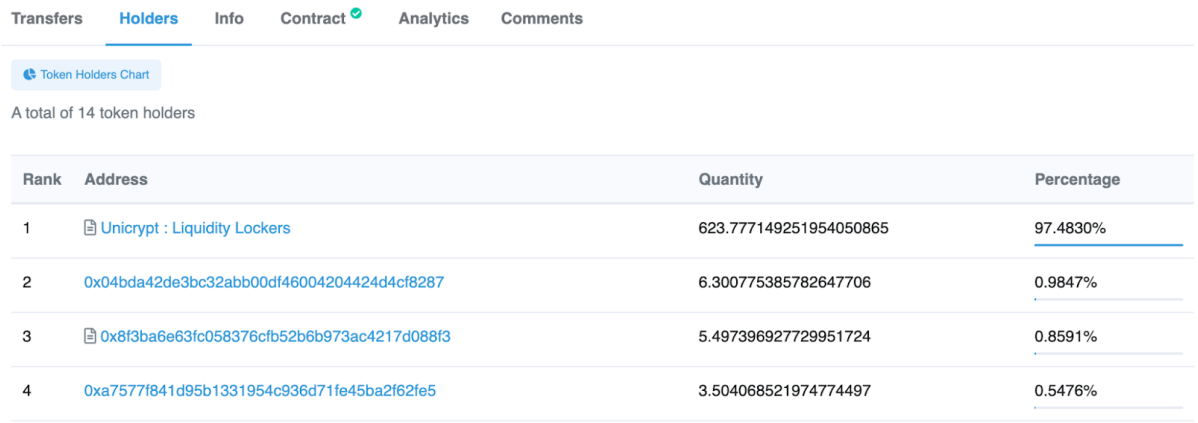
- Liquidity is locked;
- Liquidity is sufficient.

Locked liquidity means that its provider cannot remove it out of the pool until a certain period of time is over. Some projects opt for locking the liquidity they provide permanently through burning it a.k.a. sending LP tokens to the zero address so that the liquidity is always there for trouble-free token swaps.

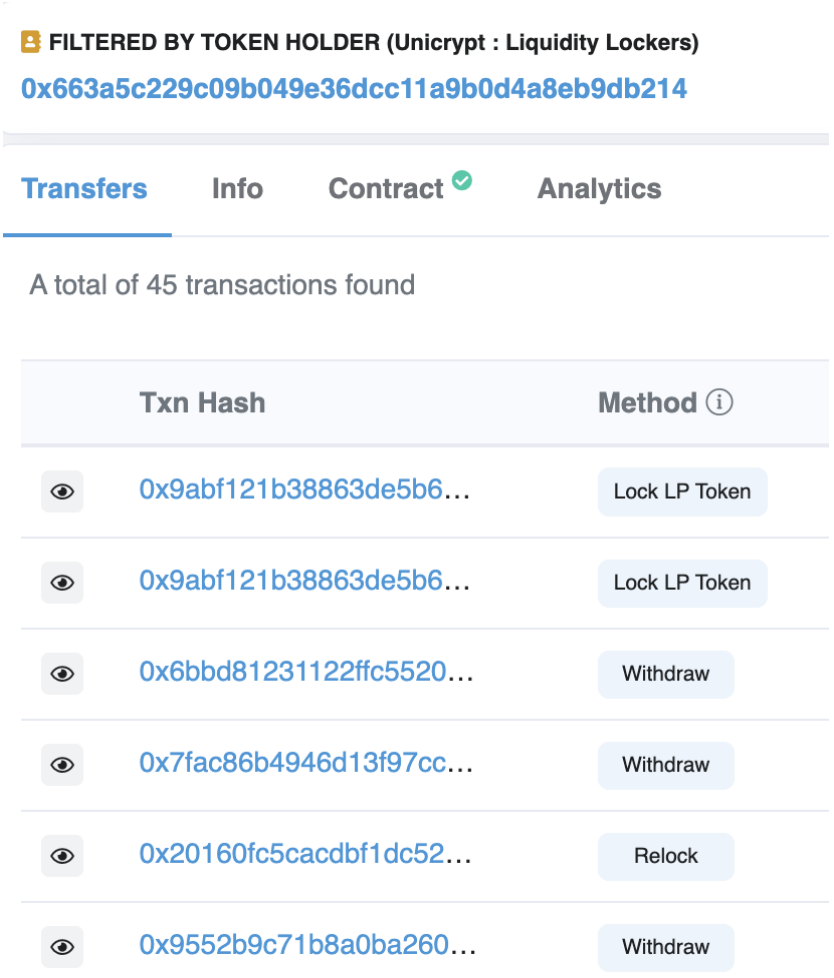
The lock guarantees liquidity cannot be pulled out of the token pair soon meaning you’ll be able to sell the token.

How exactly can liquidity be checked?

1. Open the LP token contract on Etherscan or another blockchain explorer. Addresses of all trade pairs for the token can be found either on platforms like https://dexscreener.com/ or by looking through token holders on Etherscan;
2. Navigate to the Holders section of the LP token;



3. LP tokens must be held by locking contracts or by zero address. In the example given above, we can see that 97% liquidity is locked on a Unicrypt contract;
4. If liquidity is held by a lock contract, check what is the unlock time. In order to do that, click on the contract holding LPs:



5. Open a LP locking or relocking transaction. Let's check that Relock we see on the screen.
6. You'll see the unlock time expressed in Unix timestamp in the transaction input section. Click the "Decode Input Data" button to view the input values.

#	Name	Type	Data
0	_lpToken	address	0xC70bB2736e218861DCa818d1e9f7A1930Fe61E5b
1	_index	uint256	0
2	_lockID	uint256	0
3	_unlock_date	uint256	1661940000

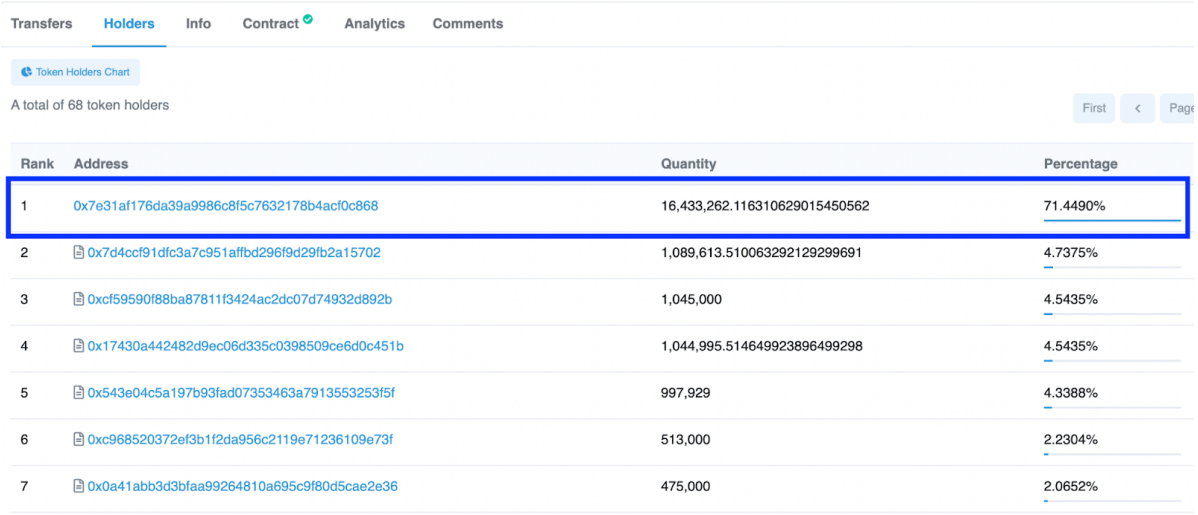
If it still sounds too complicated, no worries. You are protected by De.Fi's solutions here too. Just use our smart contract scanner when you do your due diligence and each time you are going to purchase a token. Both solutions allow you to get all the token's security information, including its liquidity state.

When it comes to evaluation if added token liquidity is sufficient, there is no single objective value to take as a reference point, but it's definitely worth avoiding pools where the project team was able to add liquidity worth a few thousand USD. Price of a token can be easily manipulated under low liquidity. This makes the token price extremely volatile.

Such token pools can attract with their impressive, temporary price surge. How easy it is to pick it up, just as easy it will be to knock it down. But how easy it can be pumped, just as easy it will be dumped.

Unfair token distribution

Here you should analyze the major token holders. Navigate to the Holders section of the token on a blockchain explorer and analyze what addresses own significant token shares:



Transfers | Holders | Info | Contract | Analytics | Comments

Token Holders Chart

A total of 68 token holders

First < Page

Rank	Address	Quantity	Percentage
1	0x7e31af176da39a9986c8f5c7632178b4acf0c868	16,433,262.116310629015450562	71.4490%
2	0x7d4ccf91dfc3a7c951affbd296f9d29fb2a15702	1,089,613.510063292129299691	4.7375%
3	0xcf59590f88ba87811f3424ac2dc07d74932d892b	1,045,000	4.5435%
4	0x17430a442482d9ec06d335c0398509ce6d0c451b	1,044,995.514649923896499298	4.5435%
5	0x543e04c5a197b93fad07353463a7913553253f5f	997,929	4.3388%
6	0xc968520372ef3b1f2da956c2119e71236109e73f	513,000	2.2304%
7	0x0a41abb3d3bfaa99264810a695c9f80d5cae2e36	475,000	2.0652%

https://bscscan.com/token/0x1C2DdB78F1fBDE389D15acd275ce6b51EAFbBA14#balances

When a private wallet or an unverified contract owns a large share of a token's total supply, there is a risk that this token amount will be dumped on an exchange. What total supply percentage is enough for dumping the token price, depends on available liquidity in pools. Owning even 5% of the token supply might be enough to cause significant volatility of the token price.

Transfer Fee

Now let's discuss transfer fees and how you can check if any fee is imposed on ERC20 transfer.

There are a few risks associated with this token feature:

1. If there is a transfer fee and it can be set to 100% without any limits or is already set to the maximum, users will lose their transferred tokens. We think the max acceptable value is 5%.
2. Even if the transfer fee is low (below 5%), it still can break functionality of a protocol the token is used in or make it vulnerable to exploits. An example is the PolyYeld exploit.

Honeypots

Another way projects can rug pull is making tokens non-transferable. Unsellable tokens are usually called Honeypots.

There are many ways to make a token a Honeypot. We'll consider the most common of them, such as blacklisting addresses, pausing token transfers and setting transfer amount limits.

Pausing

Pausing in the transfer() and transferFrom() functions can make the token non-transferrable for all users with one click. The contract admin/owner can set a state under which any function containing a requirement for this state will be blocked from execution.

Let's look at the following example:

```
function transfer(address _to, uint256 _amount)
    public
    whenNotPaused
    notBlacklisted(msg.sender)
    notBlacklisted(_to)
    returns (bool)
{
    require(_to != address(0), "can't transfer to 0x0");
    require(_amount <= balances[msg.sender], "insufficient balance");

    balances[msg.sender] = balances[msg.sender].sub(_amount);
    balances[_to] = balances[_to].add(_amount);
    emit Transfer(msg.sender, _to, _amount);
    return true;
}
```

This transfer() function contains a function modifier whenNotPaused. It contains a requirement for the state of the "pause" variable to be not paused:

```
modifier whenNotPaused() {
  require(!paused, "whenNotPaused: contract paused");
  _;
}
```

The pause state can be changed by the Pauser role.

```
function pause() public onlyPauser {
  paused = true;
  emit Pause();
}
```

You can easily read who is Pauser in the Read contract section of a blockchain explorer:

8. owner

9. symbol

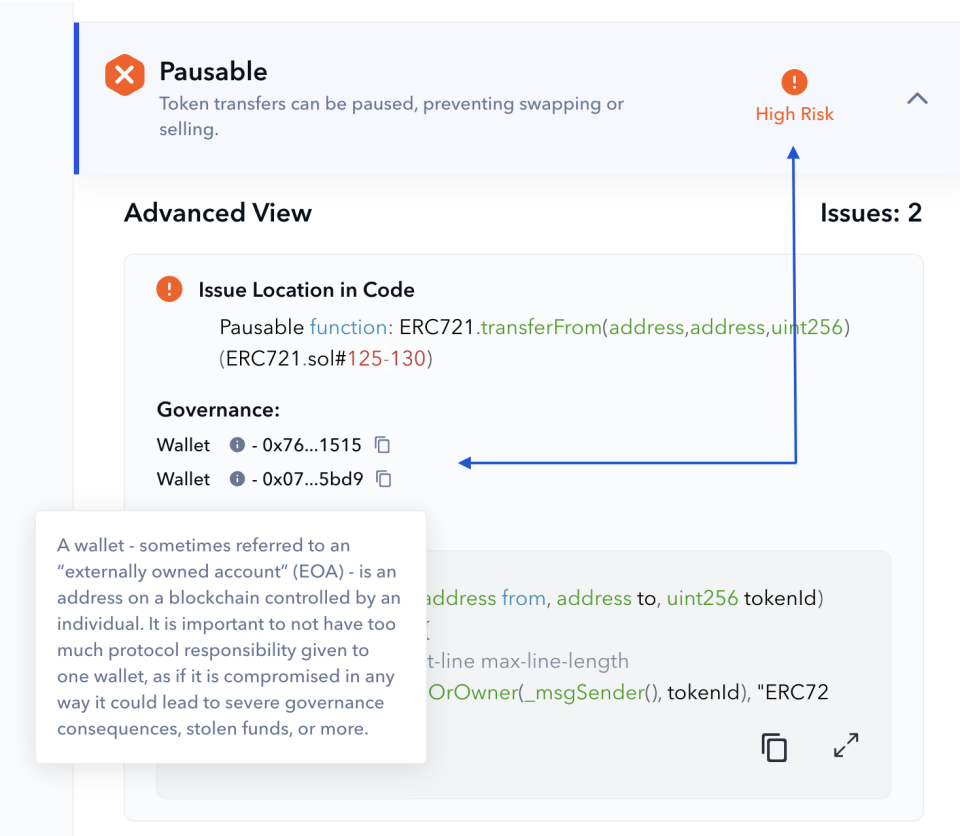
10. pauser

0x60eAfc45C80918f23e2Bd42557c55CFd069AeEc0 *address*

11. isMinter

In this case, the Pauser role is given to a regular wallet meaning the latter can call the pause() anytime completely blocking users from selling their tokens.

Let's see how the scanner helps you detect the function exposing users to the honeypot risk and define severity of the problem based on the function caller. The contract used is 0xcec4d283aa944fbce8abf6aba8d0f7af37f570f4 (Ethereum).



It's indicated that 2 wallet addresses are able to call pausing. So if they have intention to disable any token transfers and prevent the token from selling, they can do it without any limitations.

You also should know that if the approve() function, which we talked about earlier in this book, can be paused, it is another way to turn the token into a honeypot. Once approve() is paused, it is impossible to swap the token due to the fact that the way to set a DEX router to be a spender of the token is completely blocked. Every swap function on any DEX router has the transferFrom() call inside it, which requires that the swap contact (the router) has the approval from the user to move their balance.

Blacklisting

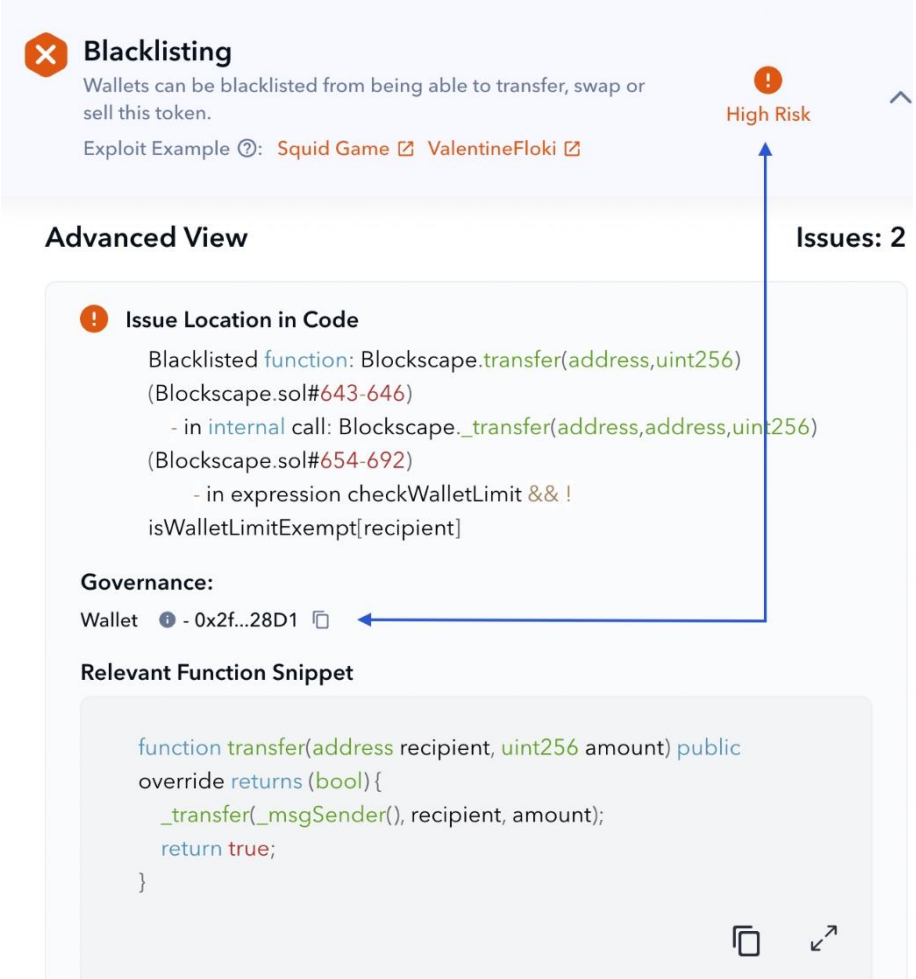
Blacklisting is a way to block users from calling a specific function through writing selected addresses into a list that is not allowed to call this

function. When we are talking about making ERC20 tokens unsellable, blacklisting in the transfer() function must be looked for.

```
function _transfer(
    address sender,
    address recipient,
    uint256 amount
) internal override whenNotPaused {
    require(balanceOf(sender) >= amount, "Not enough tokens");

    //black listed wallets are not welcome, sorry!
    require(
        !blacklistWallet[sender] && !blacklistWallet[recipient],
        "Not welcome!"
    );
```

Let's analyze another example of scanner results. The analyzed token is 0x419264d79b92b8DE3C710AB0cD3406Cd11990e02 (BSC).



The scanner's advanced view helps us to see that the possibility of transfer blockage comes from the checkWalletLimit check that contains the isWalletLimitExempt condition.

The risk is estimated as **High** due to the indicated Governance type. You can see that it is a regular wallet. Technically this means that the wallet can disable token transfers anytime making the token unsellable.

Some tokens can include blacklisting for security reasons, such as USDT. This is a clear centralization, but for legit projects it's acceptable as we know this functionality will only be called on purpose and not to rugpull users.

Transfer amount limits

Another way to control user transfers in a centralized way is setting limits on token transfer amount. Pay attention to both upper/lower limits.For example, if the maximum amount is set to 0 or the minimum amount is a very big number, transfers are blocked completely. Hence, it becomes impossible to sell the token.

Before buying any token, check if it doesn't contain any transfer blocking functionality and is sellable with our De.Fi Scanner. We integrated the checks for transfer pause, blacklisting, transfer limits and honeypots, where the latter is based on transaction simulation in order to detect if the token is sellable.

NFT exit scams

NFTs stand aside. Abandoning is the number one way to rugpull on a project here. And looking back to 2021-2022, it was not only extremely easy for scammers, but also impressively profitable. Three easy steps:

- **Promotion.** hyping around the upcoming sale. Create some impressive roadmap. No worries about feasibility. Nobody is gonna bring it to life anyways.

- **Sale mint.** No serious development skills are even needed to create an NFT contract. They literally sell air for millions dollars. The only effort required is creating some art and putting it into Metadata.
- **Disappear.**

Most of the large scams sold out within hours. And the scammers disappeared in 1-3 days.

How not to get rekt?

Team assessment is the key in avoiding empty NFT projects created to be abandoned.

Public team is the key factor to have trust for a project. Members of such teams know that they can be easily reached out and brought to justice. They also feel constant pressure for not keeping up with the declared roadmap. Thus, such NFT projects are the best to stick to.

Masterchef risks

The variety of backdoors that can be hidden in Masterchefs is infinite. Lets cover the major loopholes applied in this contract type in order to deceive users.

- Ability of a dev team to **withdraw tokens** managed by a Masterchef;
- **Pausing withdrawals** of user staked funds leaving users without any ways for an emergency withdrawal. It might be acceptable to pause any token withdrawals from a Mastechef for security purposes, for example, when the contract is breached. But under any conditions, users should be able to call emergency withdrawal, which is getting your staked funds back without carrying about earned rewards.
- Applying **deposit and withdraw fees**. Users can lose their staked tokens and gained rewards if a fee is too high. Imagine the

withdrawal fee can be set up to 100%? In this case, you won't get your invested funds back.

- Hidden, **malicious token approvals**. It's a **red flag** if there is a function in the Masterchef that contains approvals to a suspicious address when it is absolutely not needed for standard Masterchef operations. Basically, no functions besides migrate() are supposed to include token approve.
- Dangerous **LP migration**. Although it is a normal functionality used by major DeFi protocols like SuchiSwap and Pancakeswap, it can be misused in case the project applying it is a scam. This function allows migrating LP tokens from any selected pool to a new LP contract. The key is what address is set as a migrator. Is it a trusted destination? If not, all liquidity provided by users can be stolen.

Suspicious privileged functions

Regardless of the contract type and the category of the analyzed project, suspicious privileged functions are something to watch out for.

We can say a privileged smart contract function is suspicious and potentially dangerous for users when it is not needed for the contract to perform its targeted functionality and for the protocol as a whole, and at the same time it allows the contract admin, or the contract owner, or any other privileged role to take control over users' assets. That is totally unnecessary and contradicts the nature of decentralized finance.

Some smart contract developers can claim they introduce such controversial functions for security reasons. But what happens indeed is that they go beyond the promised logic. At this point this probably sounds confusing. Thus, let's look at an example.

In many contracts, you might come across the so-called **ERC20 rescue** function. It is introduced in order to get tokens, sent to the contract by mistake, out of it. In other words, it is needed to prevent random tokens from being stuck forever on the contract's balance. So dev teams add a rescue function that can only be called by admin and allow to transfer any stuck ERC20 to a selected destination. It can look like this:

```
1343
1344      // rescue tokens inadvertently sent to the contract address
1345      function ERC20Rescue(address tokenAddress, uint256 amtTokens) public {
1346          require (msg.sender == OWNER);
1347
1348          ERC20(tokenAddress).transfer(msg.sender, amtTokens);
1349      }
1350
```

A **huge problem** that you could see on the example code above is that exactly **ANY token** can be sent out of the contract balance, including the ones that are staked by users and must only belong to them! Under no circumstances should funds invested by users be directly accessible to some privileged address, because at any time this privileged address can take user funds and maliciously withdraw them out of the contract.

As the DeFi rekt history shows, similar functions are very much loved by scammers. The Compounder Finance case instantly comes to mind. The $12M+ rug pull was based exactly on the function introduced to "rescue" stuck tokens. But indeed, it was added so that the scammer's wallet, which was set to the Strategist role, could drain all 7 strategies that held invested user finds.

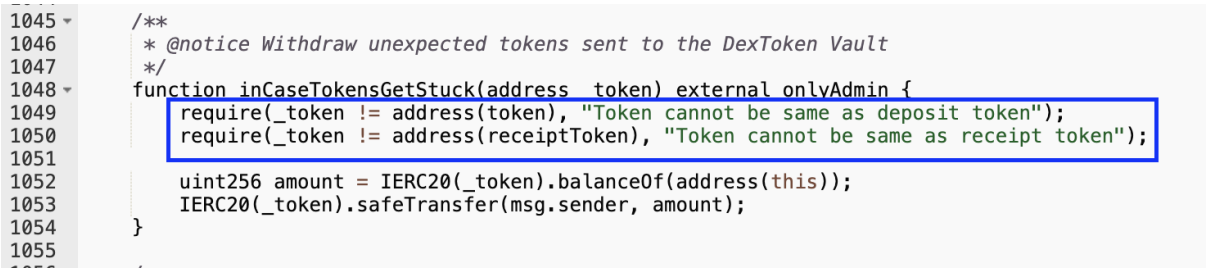
```
592
593       function inCaseTokensGetStuck(address _token, uint _amount) public {
594           require(msg.sender == strategist || msg.sender == governance, "!governance");
595           IERC20(_token).safeTransfer(msg.sender, _amount);
596       }
597
598       function inCaseStrategyTokenGetStuck(address _strategy, address _token) public {
599           require(msg.sender == strategist || msg.sender == governance, "!governance");
600           IStrategy(_strategy).withdraw(_token);
601       }
602
```

https://etherscan.io/address/0x0b283b107f70d23250f882fbfe7216c38abbd7ca#code

We highly encourage you to analyze at least few transactions when the function was called by the malicious deployer address:

- https://etherscan.io/tx/0x18e0efcaabe64299666fd78bb33dae2a4b25c6f11b469fc0498db714970cacfa
- https://etherscan.io/tx/0x9c75f70670d94e6d37f60a585f9b57d13193998d64866f720489efbea4809056
- https://etherscan.io/tx/0x0763afe207015ed7c1aa8858d2c092cf7b6a20397f2408bff20b044ef1901822
- https://etherscan.io/tx/0x10d245e61e76c7bf44257985789463ed89f624a0d5ffc45cfa671b16a7113d77

How such a high risk token rescue function had to be fixed to exclude the possibility of the centralized access to user funds ? A requirement must be added that the staking token can not be touched. Take a look at a good rescue function:



```
1045    /**
1046     * @notice Withdraw unexpected tokens sent to the DexToken Vault
1047     */
1048    function inCaseTokensGetStuck(address _token) external onlyAdmin {
1049        require(_token != address(token), "Token cannot be same as deposit token");
1050        require(_token != address(receiptToken), "Token cannot be same as receipt token");
1051
1052        uint256 amount = IERC20(_token).balanceOf(address(this));
1053        IERC20(_token).safeTransfer(msg.sender, amount);
1054    }
1055
```

This logic is totally fine and safe and meets the declared purpose of the function.

Quick Summary

Avoid:

- Sharing your wallet private key to any website and to any individual reaching you through any communication channel.
- Signing cryptographic messages you don't understand.
- Signing transactions you don't understand consequences from.

- Protocols not following industry standards, especially tokens that have basic functions but written in a custom way (without inheriting standard ERC libraries).
- Protocols that are funded through mixers.
- Projects created in 1-3 days right before the launch;
- Projects organized by teams with shady history;
- Project with no external audits;
- Projects with high level of centralization;
- Unlocked token liquidity;
- Tokens with restrictions on selling them.
- Tokens that unexpectedly got onto your wallet: they can be fake airdrops.

Remember to:

- Store your crypto on a hardware wallet;
- Review contracts that have permission to move your token balances - a.k.a. approved contacts;
- Follow technical and security updates of protocols you interact with.
- Double check domains of the websites you use. Be attentive to possible phishing attacks.
- If you doubt you interpret some contract/project info correctly due to insufficient tech knowledge, or there is some untrackable info you are lacking to make your investment decision, don't hesitate to ask community or tech support of the project;
- Stay up to date with emerging scam patterns.
- Use De.Fi safety tools —>
- Use our Scanner to get detailed security analysis for any contact: just input the contract address and see detected vulnerabilities or backdoors.
- Use our Shield to get all your approved contracts analyzed for over 80 risks.

Dictionary

DAO (Decentralized Autonomous Organization) - a way to run a crypto project that relies on smart contracts and allows holders of the governance token to make all critical decisions about development of the project.

Blockchain explorer - a web interface for researching blockchain information: blocks, transactions, events, dynamics of blockchain metrics etc.

Rug Pull - a type of DeFi scam when dev teams introduce malicious code, leave backdoors in their contracts or do other fraudulent actions in order to steal user funds and disappear.

Masterchef - contract that manages deposits to farming pools, withdrawals, rewards distribution.

Vault - a contract that allows users to deposit their assets so that they can be forwarded to yield generating strategies.

Timelock - a contract that delays execution of functions in a targeted contract.

Modifier - a reusable piece of code in a Solidity smart contract that changes behavior of functions it is applied in. For example, it can be used to automatically check if a certain condition is met prior to executing a function.

Multisig (Multisignature wallet) - is a contract that requires confirmation of a specified number of owners, called threshold, before a call can be executed.

Smart contract function - a command that enables execution of certain logic.

Privilege function - a smart contract function that can be only called by a specific role defined in the contract. Usually privilege functionality is run by the contract owner or admin. There can be multiple privileged roles in the contact, or multiple addresses can share one role.

Read functions - functions that allow users and other smart contracts to read from the state without changing it.

Write functions - state modifying commands of smart contracts.