

# Web3 security

...and how even decentralised applications  
can get *catastrophically* exploited



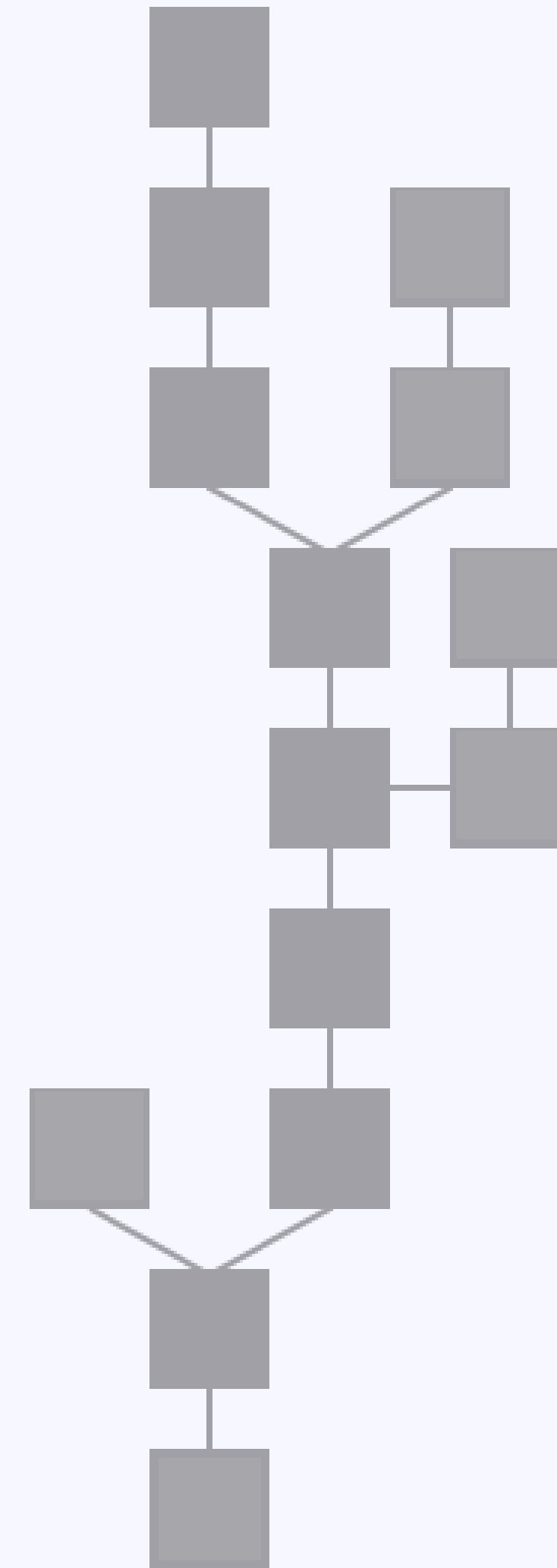
“

“The root problem with conventional currency is all the **trust** that’s required to make it work.”

—SATOSHI NAKAMOTO, CREATOR OF BITCOIN

# What we're about to see

1. What is **Web3** as a concept
2. What is a **blockchain** and what it guarantees
3. What is **Ethereum** and why it's more than just “digital money”
4. What are **smart contracts** and how they work
5. How they can (still) be **vulnerable**, despite blockchains' security measures!



# Web3 is about **ownership** and **independence**

- “Web3” is kind of a buzzword<sup>(1)</sup> used to refer to **decentralized Internet services**, where you own your data and agreements are **transparent and auditable**.
- It’s used to refer to (mostly financial) systems operating **without a central authority**, relying on **math instead of trust** for authorization and verification.
- Its core principles are **transparency** (all transactions are visible and verifiable independently), **immutability** (you cannot erase or modify a completed transaction), **interoperability** (different networks can interact with each other) and **cryptography** (remember the previous workshop?) through heavy use of asymmetrical cryptography.
- Despite the transparency, **privacy** is a huge player too (especially nowadays).

(1) **buzzword**: “A word or phrase, often an item of jargon, that is fashionable at a particular time or in a particular context.”



 IBM's article on blockchains



# **Blockchain**

(n.) A digital, decentralized, immutable ledger where transactions are stored in blocks. Once added, a block is cryptographically signed and cannot be removed, each block depends on the previous one.

Ethereum is a **computing platform** for Web3 apps.

- Ethereum itself is called “*the world’s computer*” — the idea was to have **an uncensorable, distributed, collaborative computer** that can run code, where everyone can contribute and get incentivized to do so.
- It functions as a global network that allows developers to build and run their applications, called **smart contracts**, which can be used for a wide variety of purposes.
- There’s no “central Ethereum server” — the network is maintained by a series of **independent nodes**, which work together to keep the infrastructure running.
- Its currency is called “**Ether**  $\Xi$ ” (whose shorthand is “**ETH**”), used for transactions within the network.
- Today there’s **over a million smart contracts** running on its infrastructure.

## How does Ethereum **differ** from other blockchains?

- **Bitcoin** was made for **finance** — **Ethereum** is an **applications** platform.
- **Monero** is meant for **confidential transactions** — **Ethereum's** transactions are **open**.
  - This doesn't mean Ethereum can't do the same! Platforms like **Aztec** and cryptographic proofs like **zk-SNARK** allow for confidentiality — *even in the open!*

The closest match to Ethereum is **Solana** — a more recent chain that was also built for applications and smart contracts.

YOU MIGHT NOW ASK YOURSELF...

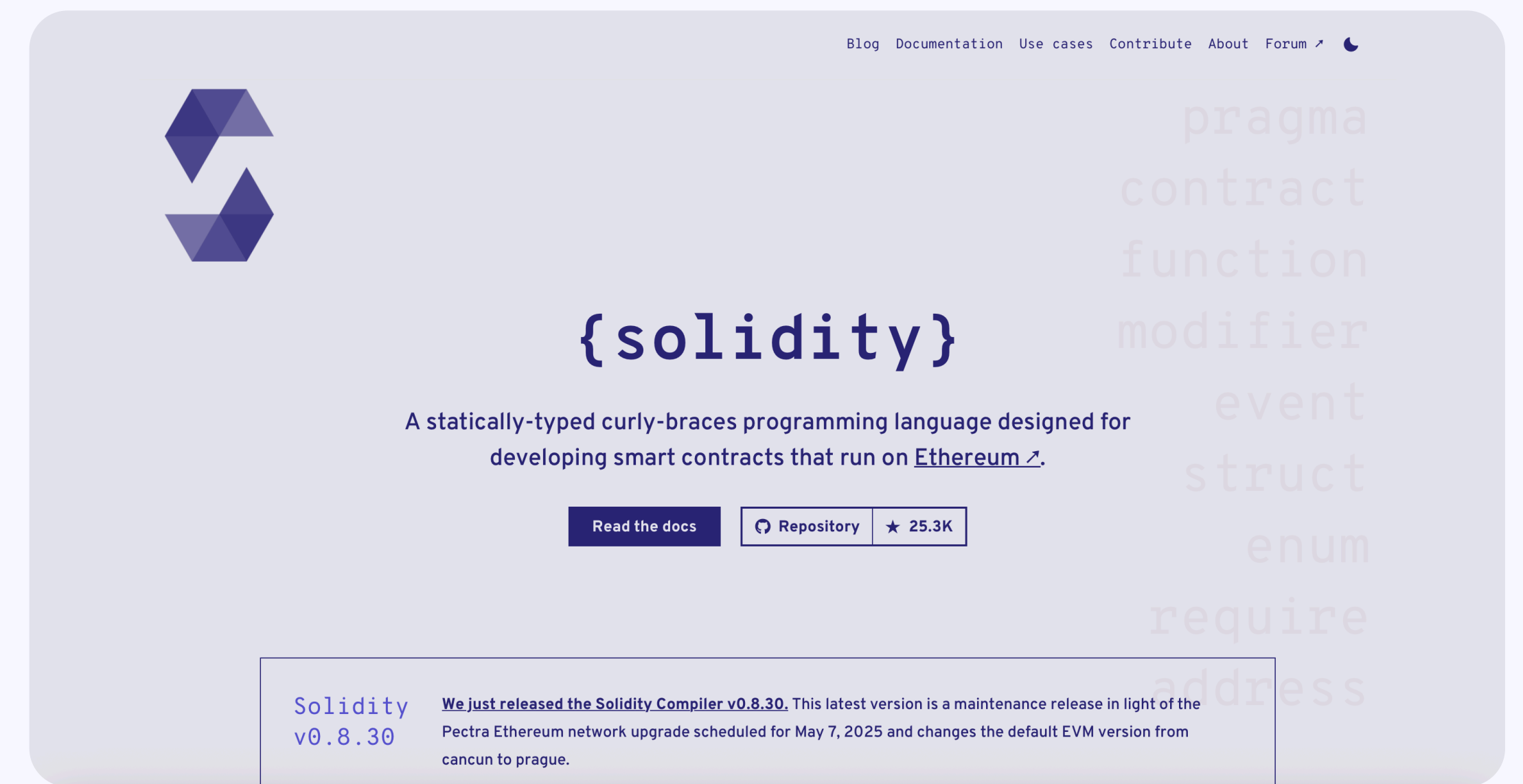
“Well, this is great and all, but...  
...how do we actually *code* an Ethereum smart  
contract?”

(P.S.: the answer is “**Solidity**”)

WHAT IS SOLIDITY?

**Solidity** is a statically-typed language designed for developing smart contracts that run on **Ethereum**.

Solidity allows anyone to write their own apps that run on the Ethereum network. It reads a bit like if the “Java” in “JavaScript” made any sense.



*Solidity's website*


## **Solidity by Example**


*Great for seeing what Solidity looks like and learning the very basics*



HelloWorld.sol	Solidity
<pre>// SPDX-License-Identifier: MIT pragma solidity ^0.8.0; contract MyContract {     function helloWorld() public pure returns (string memory) {         return "Hello, World!";     } }</pre>	

FILE EXPLORER





.deps

contracts

1\_Storage.sol

2\_Owner.sol

3\_Ballot.sol

scripts

tests

.prettierrc.json

README.txt

remix.config.json

Compile

Home

2\_Owner.sol

1\_Storage.sol

```
2
3  pragma solidity >=0.8.2 <0.9.0;
4
5  /**
6   * @title Storage
7   * @dev Store & retrieve value in a variable
8   * @custom:dev-run-script ./scripts/deploy_with_ethers.ts
9   */
10 contract Storage {
11
12     uint256 number;
13
14     /**
15      * @dev Store value in variable
16      * @param num value to store
17      */
18     function store(uint256 num) public { 22514 gas
19         number = num;
20     }
21
22     /**
23      * @dev Return value
24      * @return value of 'number'
25      */
26     function retrieve() public view returns (uint256){ 2409 gas
27         return number;
28     }
29 }
```

Explain contract

AI copilot

0

☐ Listen on all transactions

Filter with transaction hash or ad...

Welcome to Remix 1.1.3

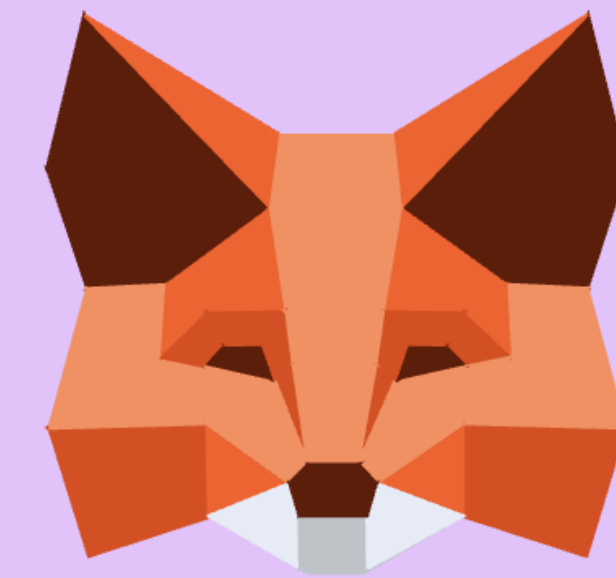
Your files are stored in indexedDB, 537.6 KB / 76.8 GB used

Let's get **set up** for  
**hacking with Ethereum!**

# 1. Get **MetaMask** and **create a wallet**

MetaMask is one of the **most popular and trusted Web3 wallets** around.

- It's available as an **extension** for all Chromium and Firefox-based browsers.
- It can also be downloaded on your mobile device as a regular **app**.



Let's get started!

Create a new wallet

I have an existing wallet

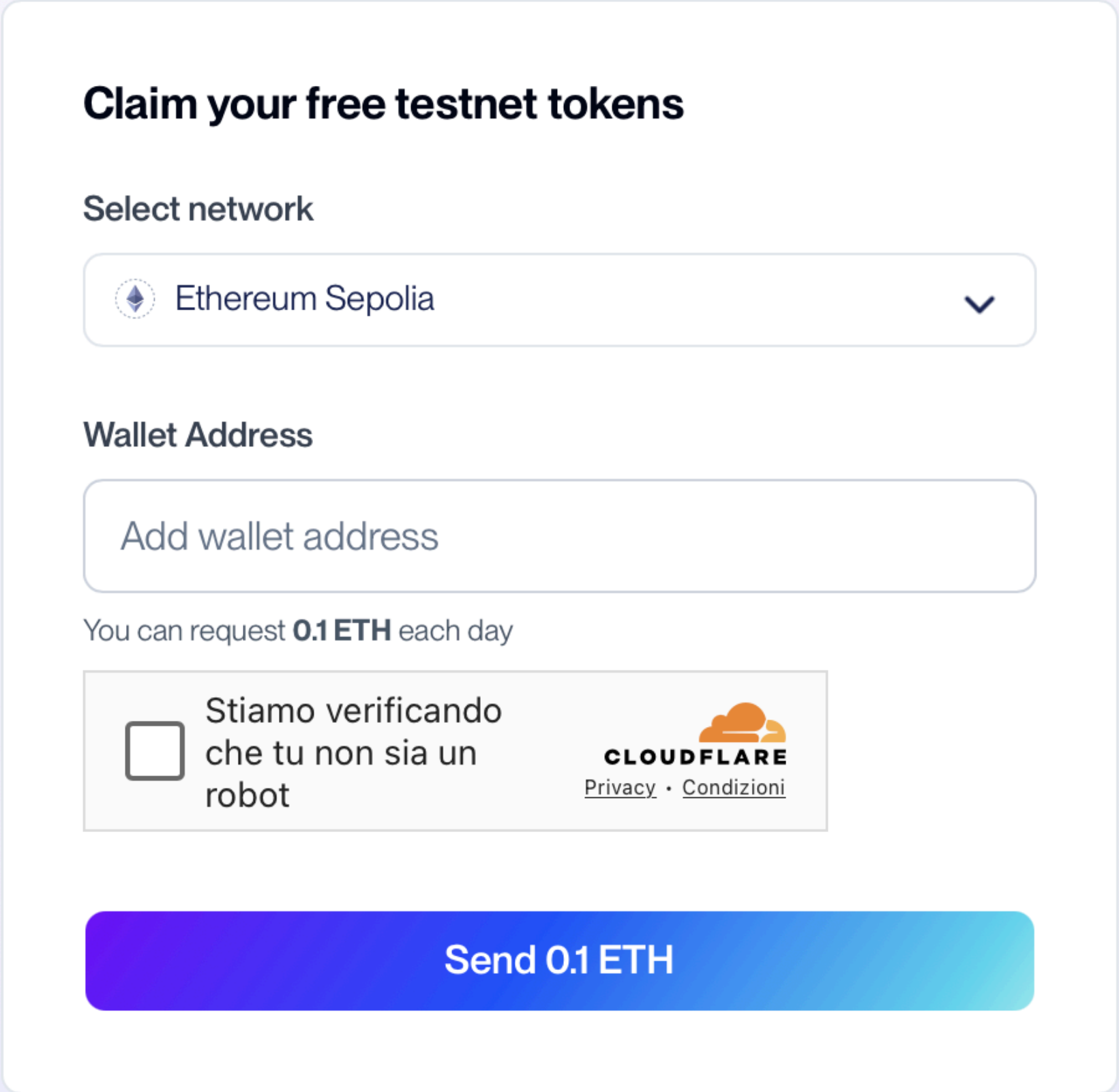


## 2. Get some **free Sepolia ETH** currency

- **Sepolia** is an **Ethereum testnet** — a development network where currency does not hold a real value. In other words, fake money. Like Monopoly cash.
- There are multiple services (called “**faucets**”) which distribute free Sepolia ETH testing tokens, so you can play around with the blockchain *without spending any real money*.
- **You will need these for trying out the challenges later!**

There are multiple free faucets available:

- [Google Cloud’s Sepolia faucet](#) (0.05 ETH / day)
- [Alchemy’s Sepolia faucet](#) (0.1 ETH / day)
- [PoW faucet](#) — mine your Sepolia



The screenshot shows a web form titled "Claim your free testnet tokens". It has a "Select network" dropdown menu with "Ethereum Sepolia" selected. Below that is a "Wallet Address" input field with the placeholder text "Add wallet address". Underneath the input field, it says "You can request 0.1 ETH each day". There is a checkbox labeled "Stiamo verificando che tu non sia un robot" (We are verifying that you are not a robot) next to the Cloudflare logo and links for "Privacy" and "Condizioni". At the bottom is a large blue button labeled "Send 0.1 ETH".

*Simply input your wallet’s address and they will send you some Sepolia ETH!*



# Don't share your **secrets**.

- **DON'T** share your *private key* with anyone!
  - There's no such thing as "Ethereum tech support". Nobody will be able to help you if your account gets stolen.
  - Fun fact: the best wallets have *multiple* private keys, with multiple layers of depth!
  - In most cases though, you will not even see your private key(s). This means...
- **DON'T** share your *seed phrase* (also known as *mnemonic*) with anyone!
  - Your recovery seed phrase is used to derive your private key(s)!
- **DON'T** upload your *seed phrase* on **any** cloud storage!
  - Don't trust companies with them.
  - Even if you do want to trust companies (why?), can you really guarantee they'll *never* suffer a data breach?
- **DON'T** store your *seed phrase* **as a file on your computer**!
  - Countless malware *specifically* targets weak wallets and files that look like they might contain seed phrases.
- **DO use a *passphrase*** to protect your *seed phrase*!
  - Decent wallets will force you to do this.
- **DO** write down your *seed phrase* with pen and paper, and store it in a place only you know!
  - Get creative.
- **DO** split your funds between accounts that you use for different purposes.
- **DO** use the 24 words option if your wallet allows for one! (Really though, it should.)

Now that we're set up,  
let's talk **vulnerabilities!**

## Vulnerability 1: **Integer Overflow** (older Solidity)

- You might know from other languages like C/C++ that in computers, **numbers have a range** they can exist within. For example, 32-bit integers span from -2,147,483,648 to 2,147,483,647. What happens when you exceed those limits?
  - Generally, it depends on the language. Some will stop at 0, others will produce a special value called NaN (**Not-a-Number**), and others will **overflow**.
- If you were to compute  $(2,147,483,647 + 1)$  in **older versions of Solidity**, the result would **overflow to the smallest possible value** and become -2,147,483,648.

This is a problem. What if **you had an allowance**, ranging from 0 to 4,294,967,295, and **set its value to 20**, then decide you want to **decrease it by 21**?

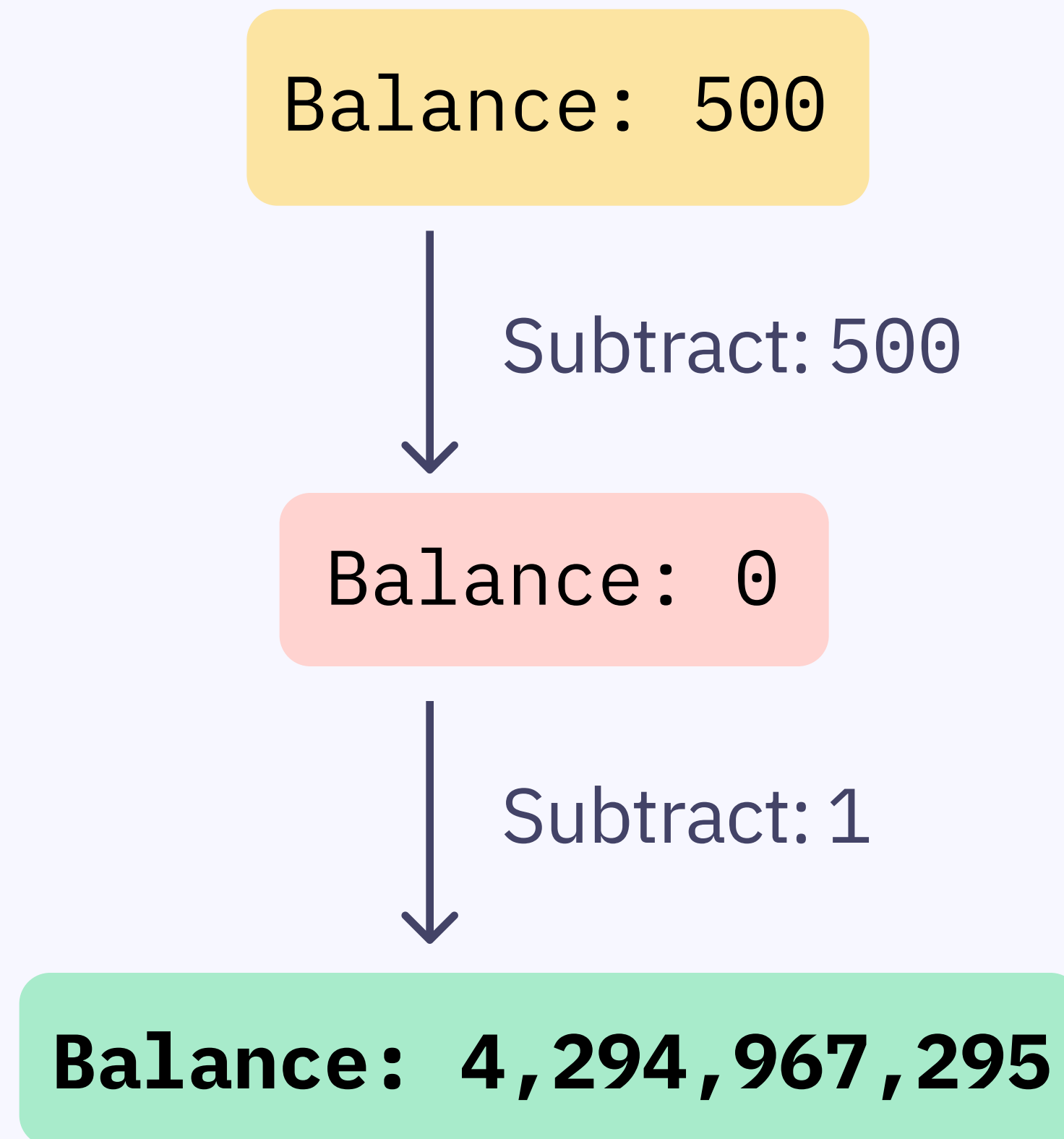
- You would end up with an allowance of **4,294,967,295!**

*not a factorial :) ↑*

## Vulnerability 1: **Integer Overflow** (older Solidity)

Upper limit: 4,294,967,295

Lower limit: 0



The balance just shot back up!

## Vulnerability 1: **Integer Overflow** mitigations

- 1. Upgrade to Solidity  $\geq$  0.8.0:** from version 0.8 onwards, Solidity automatically checks for overflow and underflow in arithmetic operations. If one were to happen, Solidity will automatically **halt the entire transaction** and all state changes get reverted back to what they were.
- 2. Use SafeMath,** a library from OpenZeppelin that automatically reverts on overflow or underflow. This would achieve the same effect — an **immediate halt upon overflow**.

**Note:** you can *still* perform unchecked overflows in newer Solidity versions too, but you must **explicitly opt-in** by surrounding your code with: `unchecked { ... }`





Link to The Ethernaut to try the exploit



Shall we **give it a shot?**

REMEMBER TO USE A BROWSER WITH METAMASK INSTALLED AND SET UP!

## Vulnerability 2: **Race conditions**

In Ethereum, **multiple users** (or other contracts) **can interact with the same smart contract at the same time.**

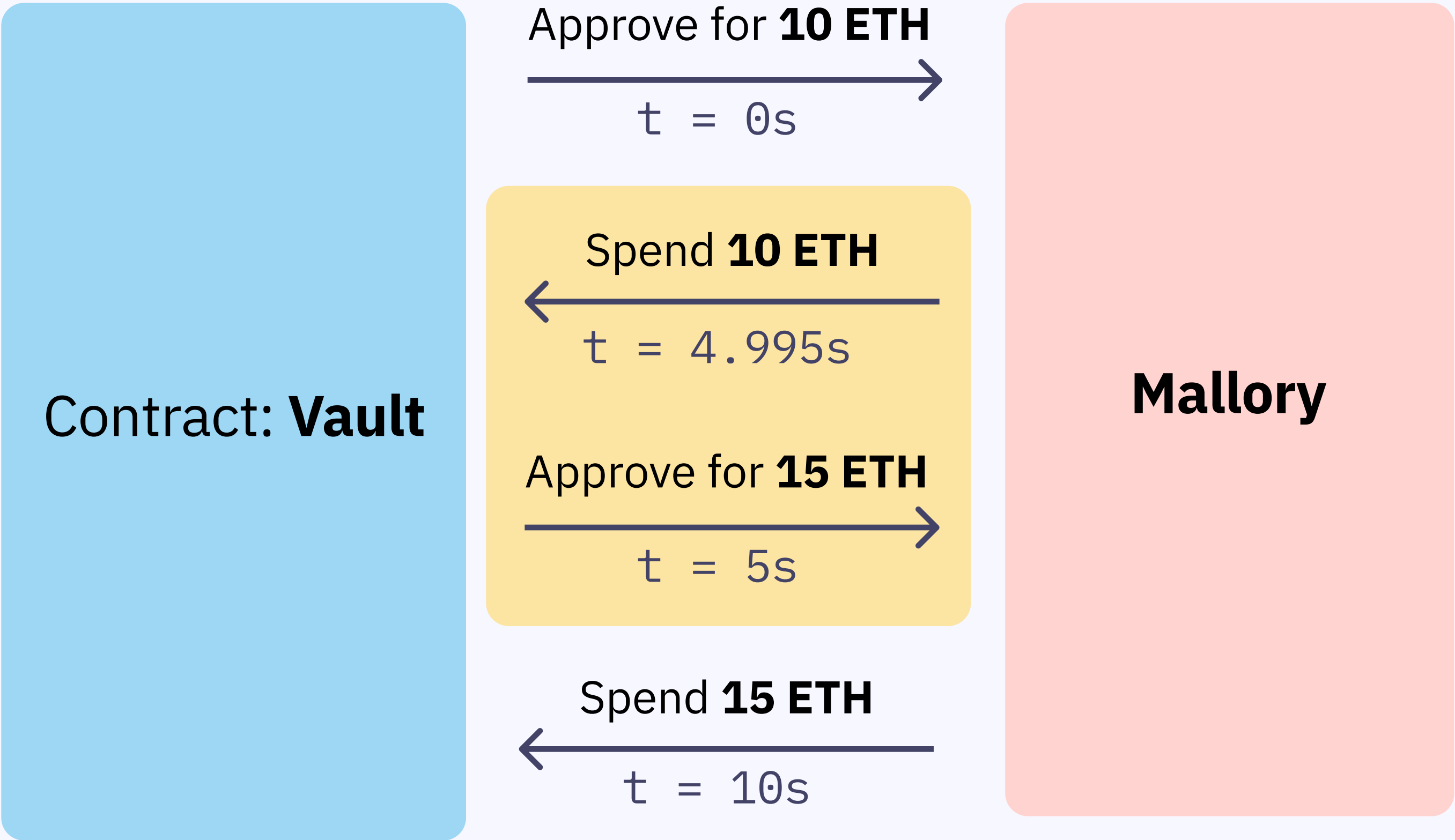
- Imagine a Token contract where you call `approve(spender, amount)` to let another address spend your tokens.
  - If you later call `approve(spender, newAmount)` while the spender is already performing a `transferFrom` using the previous allowance, both actions can overlap.

If set up precisely, the spender could use both the old and the new allowance before the second transaction takes effect — effectively spending twice what you intended.

---

*Ethereum transactions are not atomic across users; miners can include them in any order that fits the block, creating race conditions between approvals and transfers.*

# Vulnerability 2: **Race conditions**



## Vulnerability 2: **Race conditions** mitigations

- 1. Never reset an allowance directly:** instead of calling `approve(spender, newAmount)` when an allowance already exists, you should first set it to zero, or use safer functions like the built-in functions `increaseAllowance()` and `decreaseAllowance()`, part of the **ERC-20 standard**. They adjust the existing allowance incrementally, avoiding the dangerous “replace” behavior entirely.
- 2.** Some modern token implementations also use nonces or `permit()` signatures (**EIP-2612**), which let the owner sign off-chain approvals that can’t be front-run or reordered by miners.

# Logical vulnerabilities and **tx.origin** misuse



# What is `tx.origin` and how do we (mis)use it?

- `tx` is a variable holding information about the current transaction your contract is handling.
- `tx.origin` is a member that exposes the **entity who *initiated* the transaction**.
  - There's **two** main types of entity: **people** (EOA<sub>(1)</sub>) and **smart contracts**.
  - Contracts may be called by either you, or by a [**middleman**] **contract**, forming a **stack of calls**.
  - `tx.origin` will always return the *root* of the call stack.
- **Using it to verify authorization enables permission inheritance!**

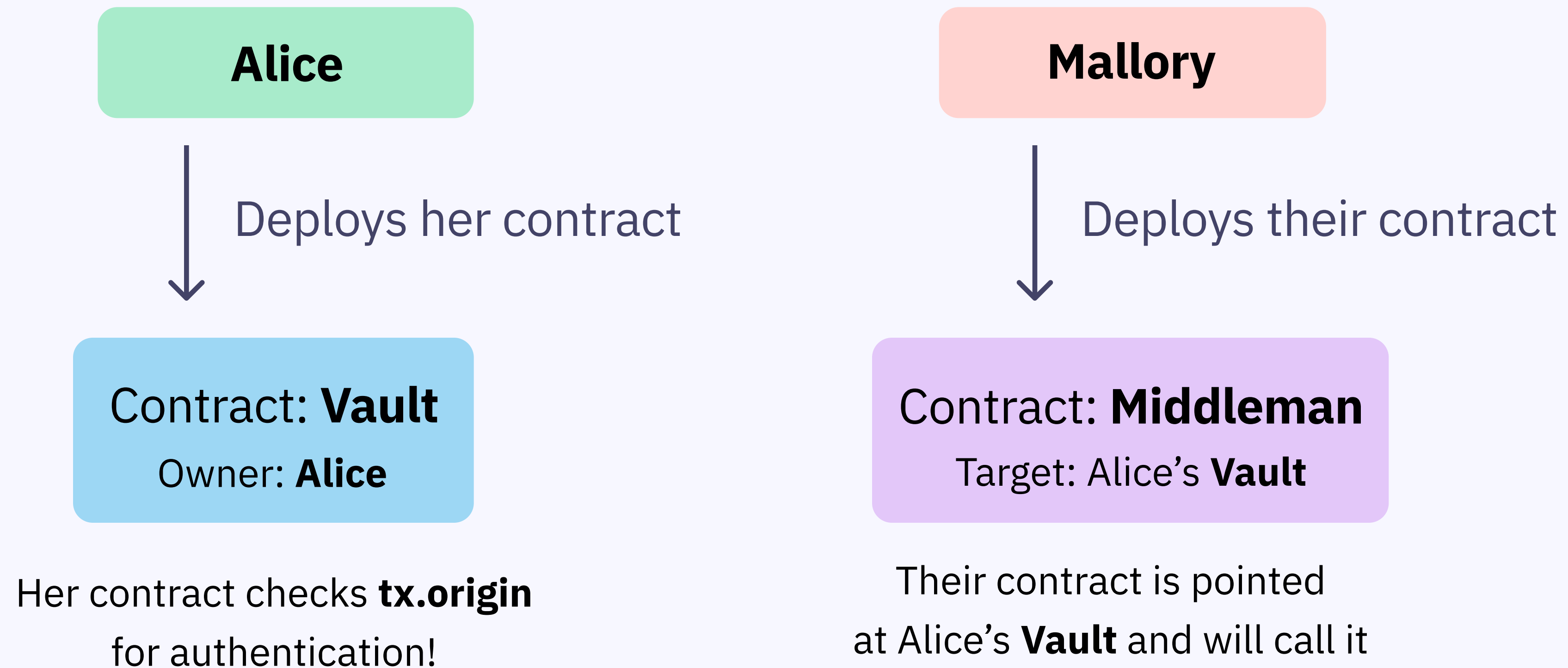
And this is exactly what we're going to do.

(1) **EOA**: Externally Owned Account, basically a person

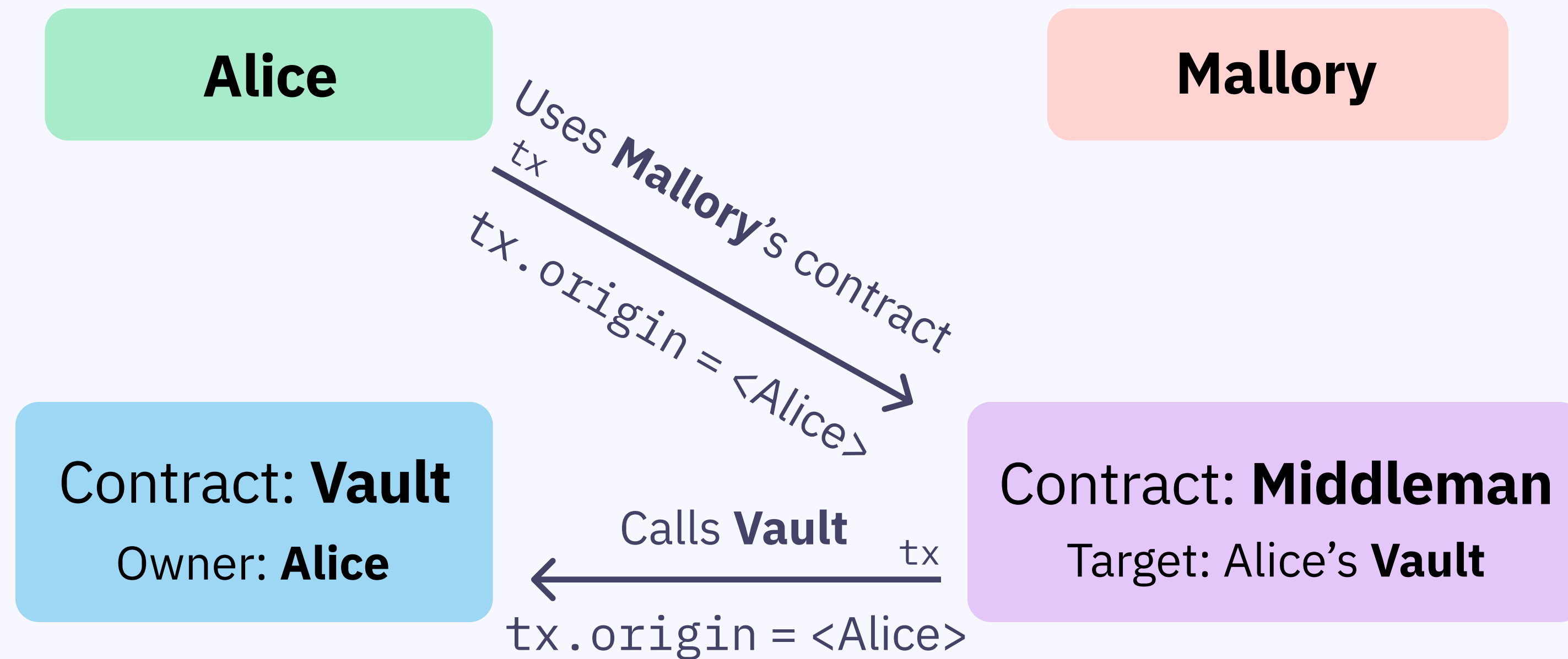
# Hypothetical scenario for exploitation

1. **Alice** is a user who deploys a contract named **Vault**.
  - As she's the one who deployed it, she also gives herself (and herself only) **admin permissions** on this contract. This allows her to call some functions nobody else should be able to.
  - She does this by storing her address in the contract as a variable, which is only set once upon initialisation.
2. **Mallory** is an evil actor who wants to obtain admin powers to do evil things.
  - Mallory makes a contract called **Middleman** and tricks Alice into using it.
  - The **Middleman** contract speaks to Alice's **Vault** and calls an **admin function**.
3. **Vault** mistakenly uses `tx.origin` to validate requests.
  - As `tx.origin = <Alice's address>`, **Middleman** (and therefore **Mallory**) just obtained **Alice's permissions**.

# tx.origin call chain (1)

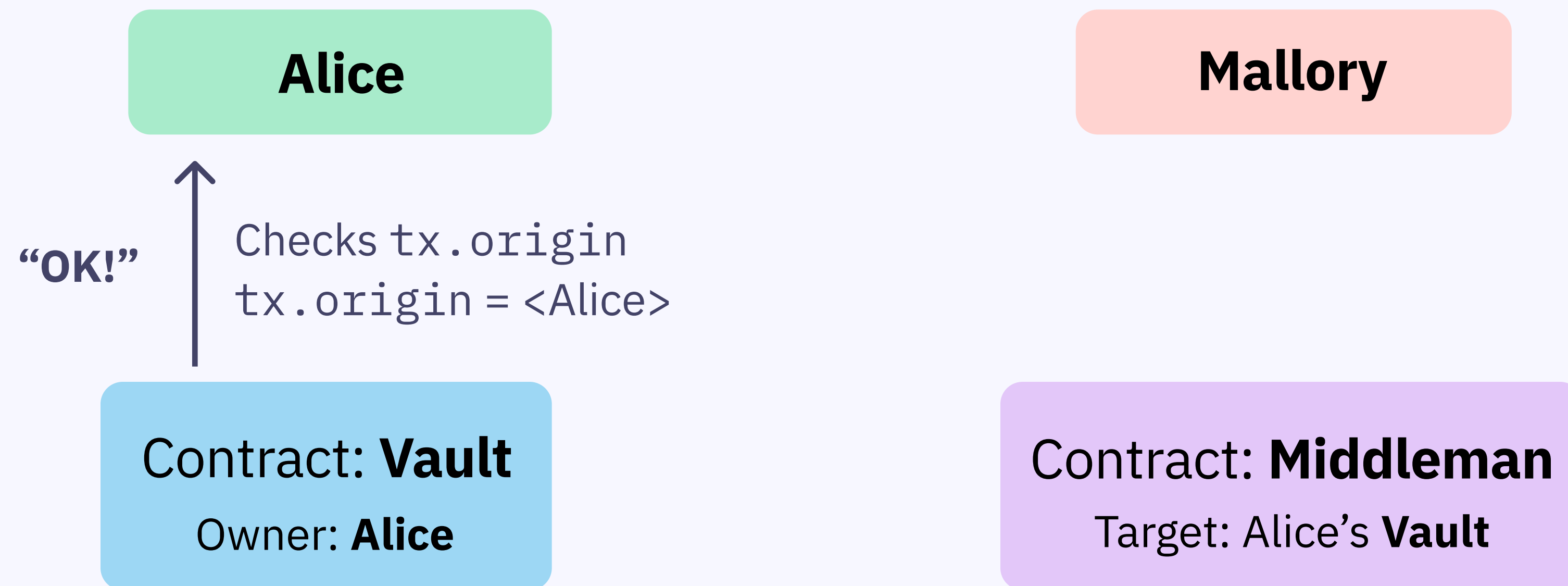


## tx.origin call chain (2)



**Middleman** has now tried  
to do something only  
Alice is authorized to do!

# tx.origin call chain (3)



Vault has now **approved** Middleman  
(therefore Mallory) to act on behalf of Alice!



Link to the GitHub repo to follow along



So **let's demonstrate that!**

START UP REMIX IDE & FOUNDRY AND GIVE IT A SHOT!

 Link to the OliCyber website  
↓

Now it's your turn! Try to solve **this challenge**.

DON'T WORRY—THERE'S NO TIME LIMIT.



# Re-entrancy attack, also known as *The DAO hack*

THIS ATTACK *LITERALLY* SPLIT ETHEREUM IN HALF!

# What does *re-entrancy* mean, and how does it work?

- When a smart contract sends Ether to another address, it does so by triggering that address's fallback or receive function — that code will then start running before the first contract finishes its execution.
- If the receiving contract is malicious, it can use this opportunity to call back into the first, vulnerable contract before the original function is done — effectively *re-entering* it while its internal state has not yet fully finished updating.

*Let's imagine you had a `withdraw(...)` function in a contract.*

*If your contract first hands out the Ether (by using `msg.sender.call`), and only updates its internal balance afterwards, **the remote party can re-enter your contract** and keep calling the function, effectively stealing all of your contract's Ether balance!*

# How can we avoid *re-entrancy* attacks?

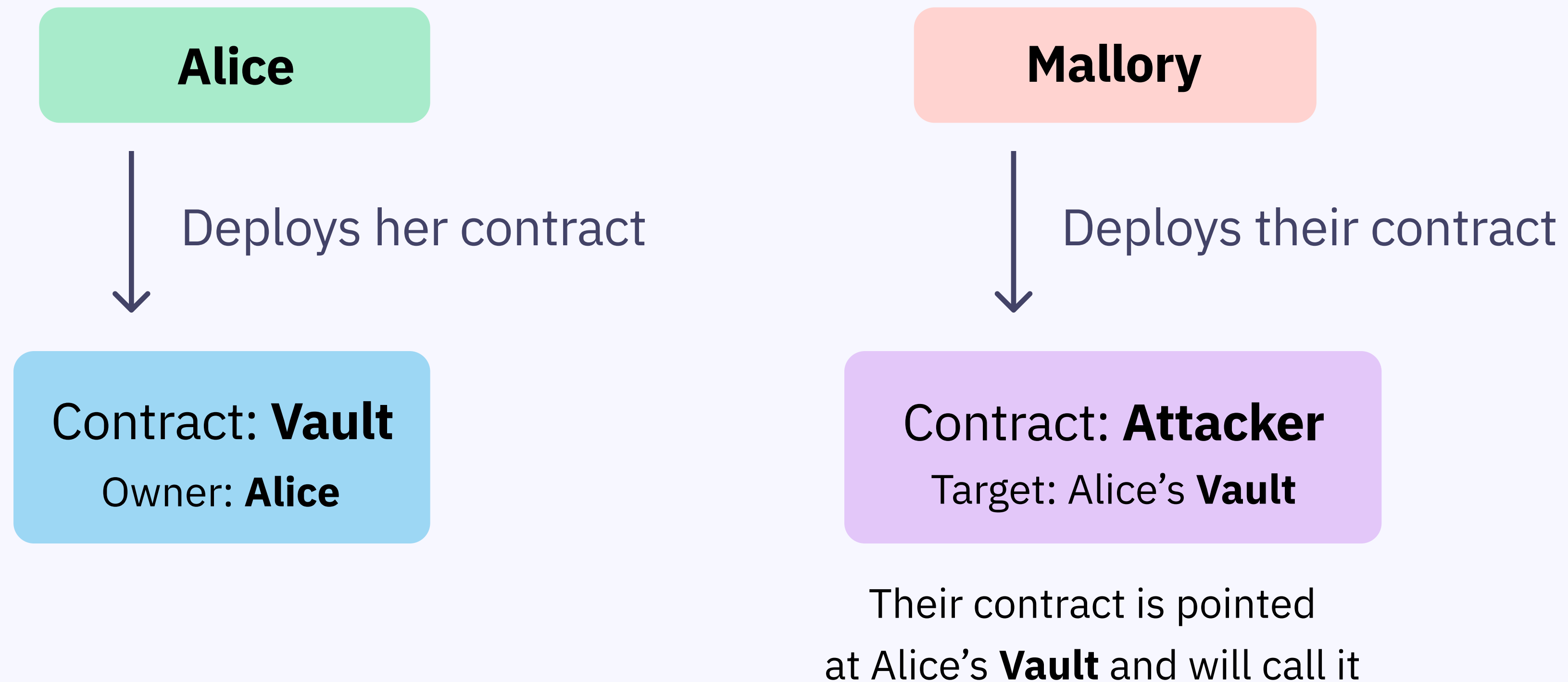
### **1. Follow the "Checks–Effects–Interactions" rule:**

- a. Check all conditions first (`require()` statements).
- b. Update your contract's state all the way through.
- c. Only then interact with external contracts or send Ether.

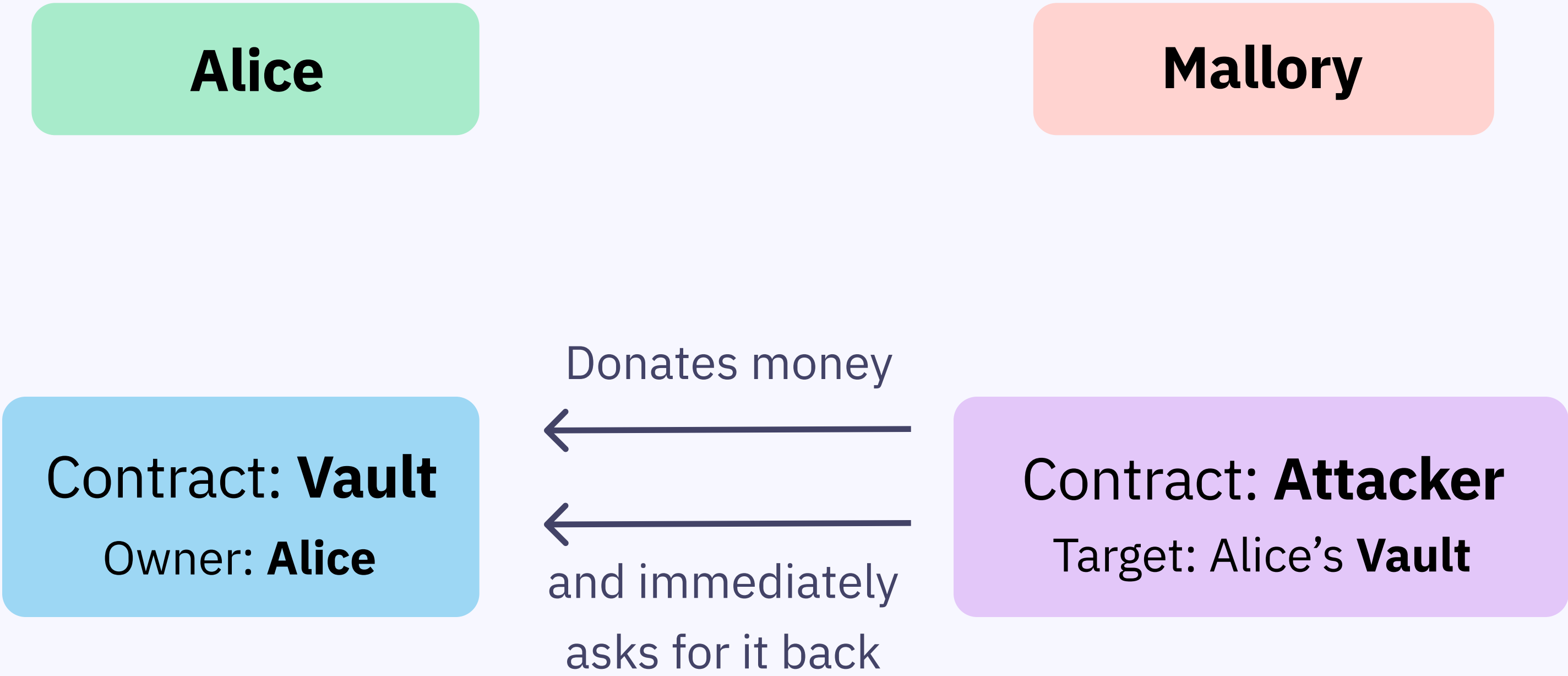
This way, even if the receiver tries to re-enter, their balance is already set to zero by the time they receive any call from your contract.

### **2. Use a reentrancy guard**, such as *OpenZeppelin's ReentrancyGuard*, which locks a function while it's running so it can't be called again until it finishes.

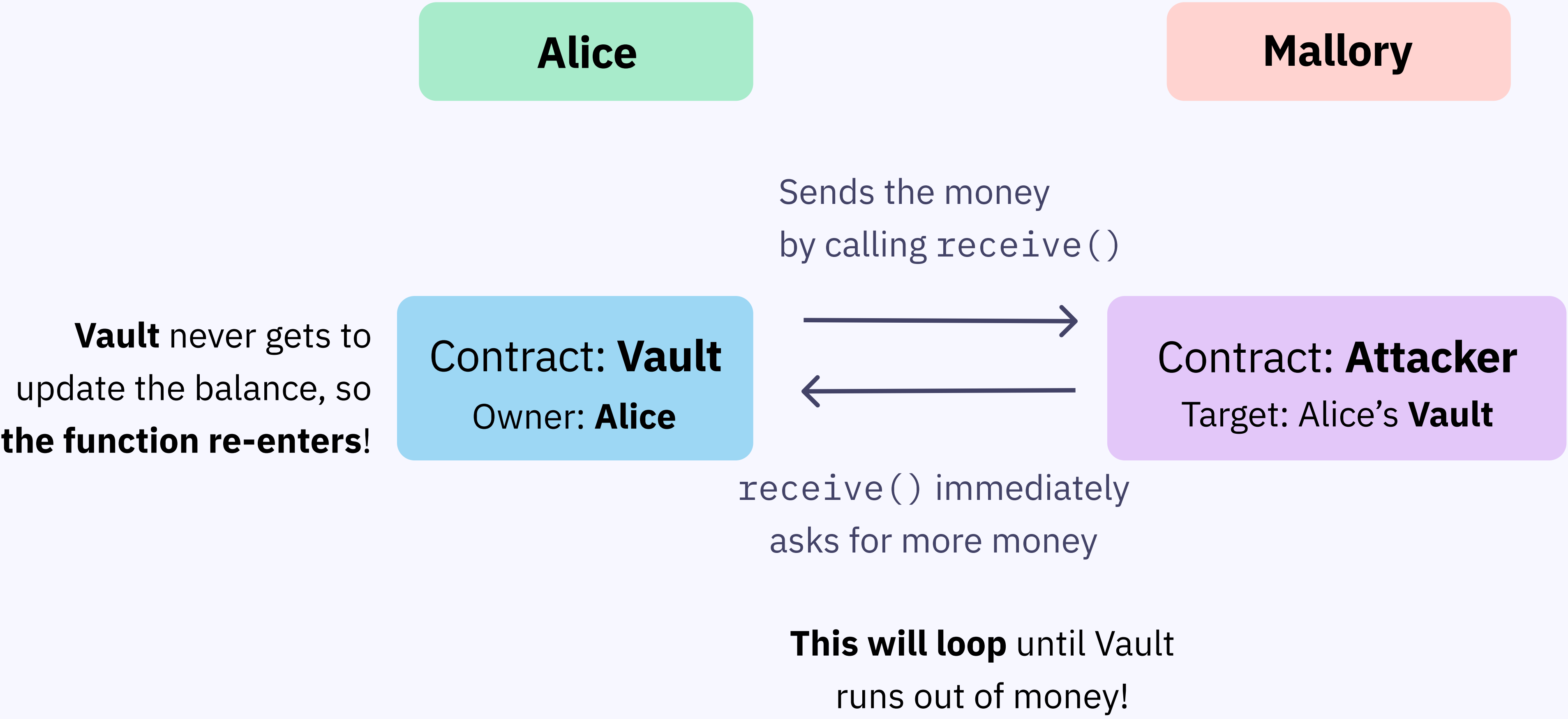
# Reentrancy call chain (1)



# Reentrancy call chain (2)



# Reentrancy call chain (3)





Link to the GitHub repo to follow along



Alright! **Let's demonstrate that** once more!

START UP REMIX IDE & FOUNDRY AND GIVE IT A SHOT!



 [Link to The Ethernaut](#)  
↓

And now you try! Try to solve **this challenge**.

DON'T WORRY—AGAIN, THERE'S NO TIME LIMIT.

# **Extra:** *re-entrancy*, immutability and ethics

LET'S COOL OFF A LITTLE AND DISCUSS SOME PHILOSOPHY!

# The DAO hack, and its consequences

- The last vulnerability we've seen, *re-entrancy*, has caused one of the most catastrophic events in the Ethereum blockchain's history — yet, no contract was violated.
- In June 2016, hackers managed to violate the largest decentralised Ethereum organisation of the time: “The DAO”. By exploiting re-entrancy, they were able to steal the equivalent of \$60 million USD in Ether, gathered through crowdfunded investing.
- This event sparked what is perhaps the most controversial decision ever taken by the Ethereum project — **rewriting history**.
  - The developers proposed a hard fork of the blockchain — *reverting the hack and returning the money to the people it was stolen from*.
  - This proposal was accepted by the majority of the community, miners and exchanges, and thus Ethereum was split into two: Ethereum and Ethereum Classic — made by the people who refused this idea.

# Should history be ever rewritten?

- If blockchains are supposed to be immutable, should we intervene when unexpected things happen?
- If smart contracts are supposed to replace the law with undisputable math and logic, should accidental oversights be considered *fair game*?
- Who decides what is the right thing to do: the code or the people?

This fundamentally redefined what Ethereum really has become:

- No longer a mere cold, mechanical technology
- But a **collective platform** where **social consensus** gets priority

And that is exactly the philosophy behind Web3!

# Thank you!

*See you on the 21st!*

## **Advanced vulnerabilities you can continue studying:**

- Denial of Service by exceeding gas
- Timestamp manipulation (“Riskless Uncle Maker”)
- Flawed DIY PRNG implementation
- Short Address Attack

## **Sources:**

- [Mastering Ethereum](#)
- [Fundamentals of Smart Contract Security](#)
- [Security Vulnerabilities in Ethereum Smart Contracts](#)

*If this world interests you, remember to check out **The Ethernaut**,  
a game where the goal is attacking smart contracts!*

Made with 💜 by [zeph](#).