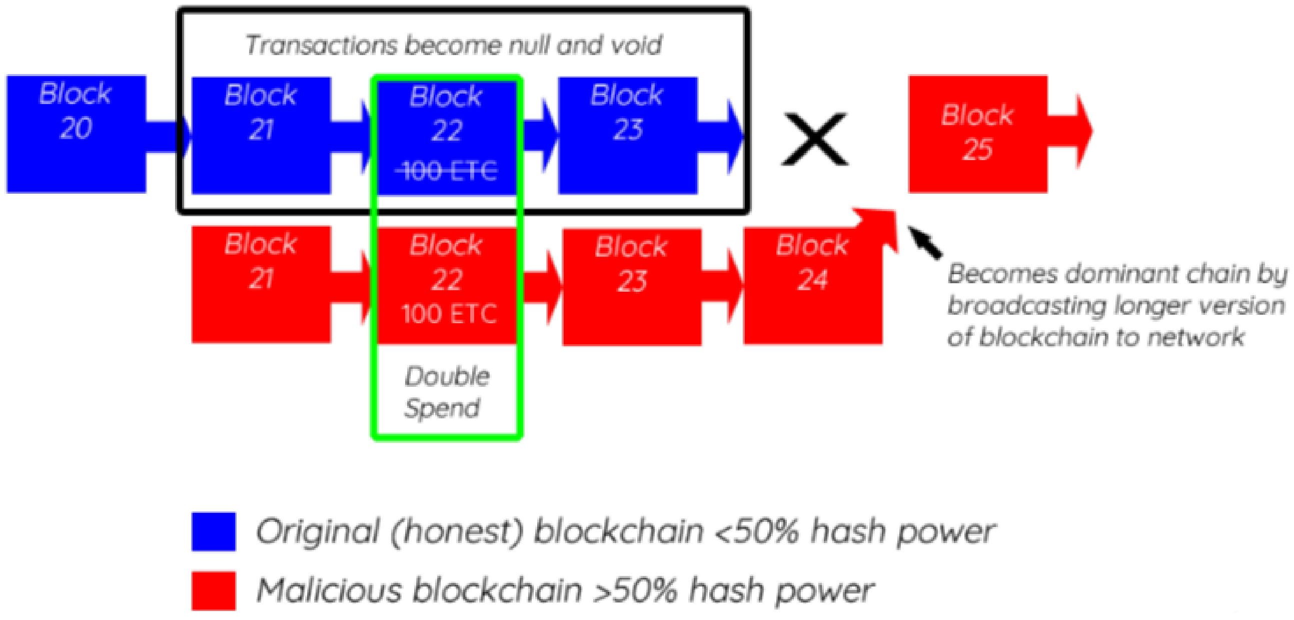
# Ethereum Security 1

## (Consensus) Protocol

Consensus attacks exploit the incentive structure of blockchains, these sort of attacks can be quantified and many papers have been written analyzing and predicting economic incentives for acting maliciously against the protocol.

### 51% Attacks

It may be feasible in certain blockchains where the incentive of rewriting the last block(s) is greater than the cost of carrying out a 51% attack.
Poor decentralization can lead to miner's collusion in carrying out such an attack, it may also be feasible to rent the necessary compute power from a VPS provider for a very short period of time needed. Ethereum Classic has been a victim of such attack thrice already.


Transactions become null and void
Block 20
Block 21
Block 22 ~~100 ETC~~
Block 23
X
Block 25
Block 21
Block 22 100 ETC
Block 23
Block 24
Becomes dominant chain by broadcasting longer version of blockchain to network
Double Spend
Original (honest) blockchain <50% hash power
Malicious blockchain >50% hash power

### MEV (Miner Extractable Value)

This attack is a lot easier to execute and doesn't require reorganizing blocks, although the profits for the cheating miner may be equally great as a 51%.

It is not a one-off attack like 51%, but rather an ongoing activity of miners inspecting the content of all incoming transactions, trying to figure out whether they'd be profitable to execute, then copying the exact bytecode from those transactions and executing it themselves whilst excluding original transaction from the mined block.
Although they will not receive gas fees, the profit made from stealing the transaction is much

greater (MEV = net profit - foregone tx fee); most often these come from arbitrage bots trying to make an honest buck, or it could also be a tx from someone calling an unprotected “withdraw” function from a contract without realizing that malicious users could notice the withdrawal and call the withdraw function themselves.

This attack is called **frontrunning**, meaning that anyone can replay the tx for themselves if they noticed it, leading to a proper auction for who pays the most gas to get the tx mined before everyone else, although miners will always have the final word as they have the ultimate power to include transactions in a block.

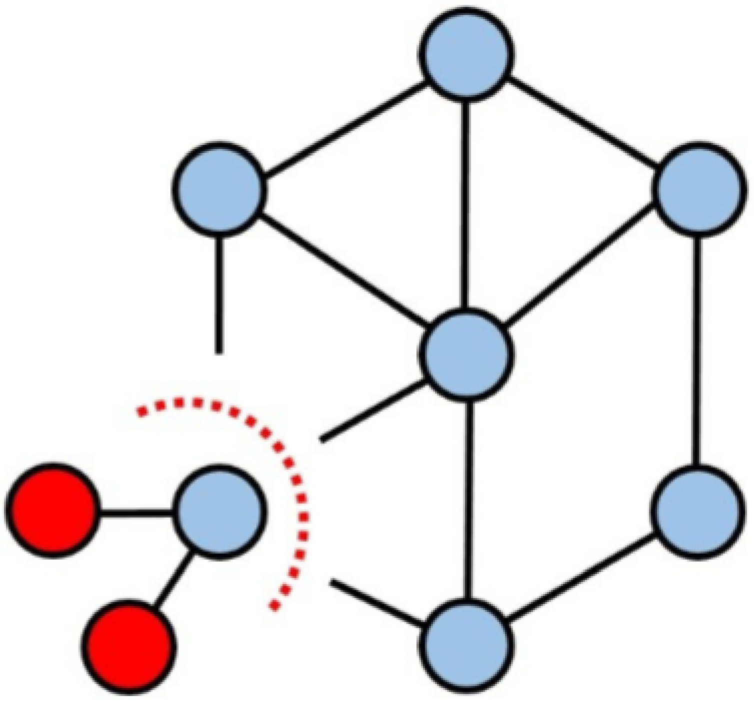
### Selfish Mining

Involves miners colluding to secretly mine a series of blocks in a row and then reveal them only when other miners catch up to being just a block behind.
In doing so, the selfish miner or pool will fool other miners who, thinking they have the longest chain, will waste computing power trying to solve a block that has already been solved; while they are busy solving this, the cheating miner can keep working on her own longest chain while everyone else is keeping busy and wasting time, thus gaining an unfair advange over the rest of the network, and finally claiming all the rewards for the mined blocks.
I.e. if a block is revealed immediately after finding the solution like honest miners do, the rest of the miners can move on and stop wasting time on an already solved block, but if kept secret, honest miners continue working on it while dishonest miner’s advantage will keep compounding block after block.

## (P2P) Network

These attacks happen at the network layer, exploiting well known vulnerabilities inherent to how computer networks are built, because blockchains need a P2P network of nodes to function, these nodes become a primary target.

### Eclipse



Blockchain nodes need to communicate with other peers in order to exchange information; in this case attackers deploy nodes themselves trying to gain the trust of the target node, slowly replacing all peers of the target node one by one. When this happens, the eclipsed node will only see what the attacker wants them to see, like that a payment to their account has been successful when in reality it was never broadcasted to the real network in the first place.

## Sybil Nodes

The Sybil attack in computer security is an attack wherein a reputation system is subverted by creating multiple identities. Some blockchain's consensus mechanism may be vulnerable to this attack, for example where the miner isn't chosen by PoW but on a random selection; increasing the number of nodes the attacker increases the chance of being selected as a miner, if the cost of running nodes is trivial and there is no slashing mechanism for acting maliciously, then Sybil nodes can undermine the whole blockchain.

## DDoS

Blockchain nodes are just regular servers, they can be overloaded by spamming them with requests exhausting all their bandwith and therefore rendering them useless.
It could be in order to attack competitor miners, eclipse a specific node from the network, or to attack a blockchain as a whole without necessarily a reason. Blockahins have bootstrap nodes IP addrasses hardcoded for new joiners, if those were targeted it could prevent new nodes from joinng the network.

For example, some blockchain clients need to return massive amounts of data in response to a single get request, this is clearly disproportionate and it means that a single laptop could perhaps bring down a node by requesting blocks at an unsustainable rate.

## Smart Contracts

### Access Control

Access Control issues are common in all programs, not just smart contracts. In fact, it's number 5 on the OWASP (Open Web Application Security Project) top 10. One usually accesses a contract's functionality through its public or external functions. While insecure visibility settings give attackers straightforward ways to access a contract's private values or logic, access control bypasses are sometimes more subtle. These vulnerabilities can occur when contracts use the deprecated `tx.origin` to validate callers, handle large authorization logic with lengthy require and make reckless use of `delegatecall` in proxy libraries or proxy contracts.

Access control was recently exploited in a trivial hack of Punk Protocol (https://www.rekt.news/punkprotocol-rekt/), where developers created a function called `initialize()` , which replaces the standard `constructor()` function in upgradable contracts; this function is always used with a modifier called `initializer` which prevents this function to be called twice. However Punk's developers did not apply the modifier, hence the hacker could call the function again after the developers, replacing the address of the proxy "admin" contract with an arbitrary address, enabling them to withraw all the funds (10m) from the protocol. So much for security…

## Centralized Points of Failure

### Protocol Administration and Governance

One of the biggest contributors to centralization risk in DeFi protocols is the use of admin keys. Admin keys allow protocol developers to change different parameters of their smart contract systems like oracles, interest rates and potentially more, these private keys controlling a smart contract's governance could get hacked or misused in ways such as draining a lending pool from its funds or freezing the smart contract rendering it useless.

#### Compound

For example, the lending platform Compound is a custodial system, all lending pools can be trivially drained if their admin private key is compromised; during a security review in 2019 of the Compund smart contracts, samczsun (https://samczsun.com/) found that Compound v2 has four different administrative positions which are set to three distinct addresses, if any of those keys would have been compromised, an attacker could have drained all of the cTokens from the Compound lending tool, among other attacks such as preventing minting and transfering of cTokens.

#### Liquid Network

Another recent example involves the Liquid network, which as pointed out on Twitter (https://twitter.com/_prestwich/status/1276300994989572096) by James Prestwich, for just under and hour, three administrative private keys controlled 870 Bitcoin due to a timelock issue; this violates liquid’s security model.

### Oracle risk

Another large element of centralization risk in these protocols is oracle centralization. Oracles provide data to smart contracts that is then used in the execution of functions. Price feeds are one of the critical pieces of infrastructure in decentralized finance and their failure or exploitation can lead to negative outcomes for users and platforms.

DeFi project, bZx, famously suffered from an oracle attack with the hackers stealing some 630,000 ETH. Chainlink and other decentralised oracle networks are helping to mitigate this risk.

Please see our medium article Price Oracle Manipulation (https://extropy-io.medium.com/price-oracle-manipulation-d46fd413cc17)

## Digression about Function Selectors

How does the EVM call the correct function in a contract ?

### Function Selector

The first four bytes of the call data for a function call specifies the function to be called.

The compiler creates something like

```
method_id = first 4 bytes of msg.data
if method_id == 0x25d8dcf2 jump to 0x11
if method_id == 0xaabbccdd jump to 0x22
if method_id == 0xffaaccee jump to 0x33
other code
0x11:
code for function with method id 0x25d8dcf2
0x22:
code for another function
0x33:
code for another function
```

## Encoding the function signatures and parameters

Example (https://docs.soliditylang.org/en/v0.6.2/abi-spec.html?highlight=selector#examples)

```
pragma solidity ^0.8.0;

contract MyContract {

  Foo otherContract;


  function callOtherContract() public view returns (bool){
      bool answer = otherContract.baz(69,true);
      return answer;
  }
}


contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) {
    r = x > 32 || y;
    }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

The way the call is actually made involves encoding the function selector and parameters

If we wanted to call ***baz*** with the parameters **69** and **true** , we would pass 68 bytes total, which can be broken down into:

1. the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature baz(uint32,bool).

   ***0xcdcd77c0:***

2. the first parameter, a uint32 value 69 padded to 32 bytes

   ***0x0000000000000000000000000000000000000000000000000000000000000045:***

3. the second parameter - boolean true, padded to 32 bytes

   ***0x0000000000000000000000000000000000000000000000000000000000000001:***

In total

***0xcdcd77c000000000000000000000000000000000000000000
000000000000000000000450000000000000000000000000000
000000000000000000000000000000000001***

This is what you see in block explorers if you look at the inputs to functions

There are helper methods to put this together for you

```
abi.encodeWithSignature("baz(uint32, boolean)", 69, true);
```

Alternatively you can then call functions in external contracts on a low level way via

```
bytes memory payload =
abi.encodeWithSignature("baz(uint32, boolean)", 69, true);

(bool success, bytes memory returnData) =
address(contractAddress).call(payload);

require(success);
```

## Fallback Function

A contract can have at most one fallback function, declared using fallback () external [payable] (without the function keyword). This function

- cannot have arguments,
- cannot return anything and
- must have external visibility.

It is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no receive Ether function. The fallback function always receives data, but in order to also receive Ether it must be marked payable.

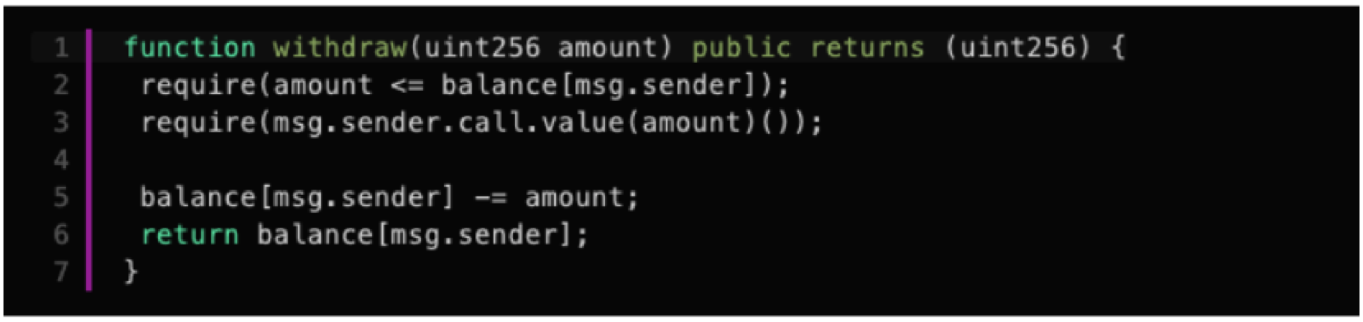
## Re Entrancy

### Vulnerable Contract

This contract stores the amount ETH an address is entitled to in

```
mapping(address=>uint256) balances;
```

and allows the address to withdraw an amount, up to their balance by calling the withdraw function

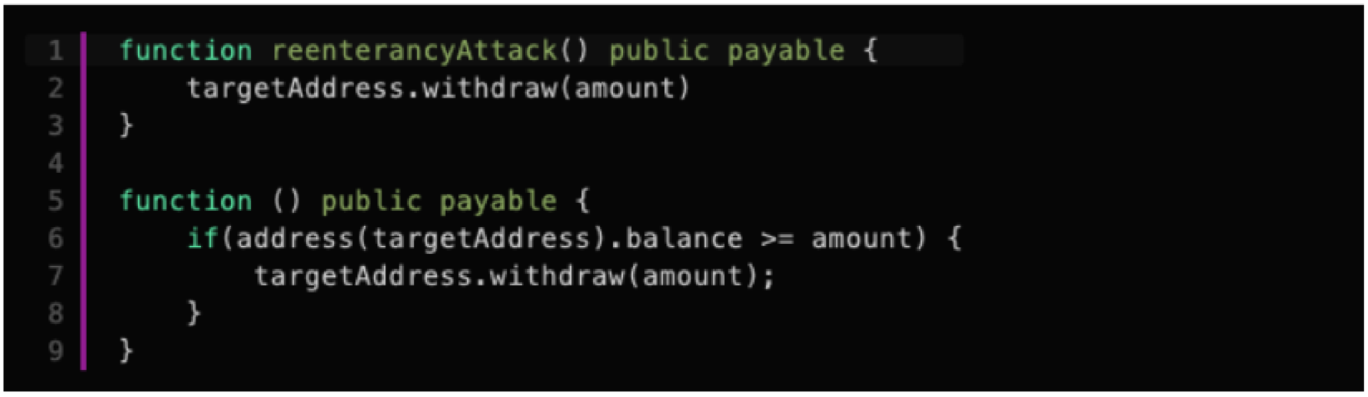


```
function withdraw(uint256 amount) public returns (uint256) {
 require(amount <= balance[msg.sender]);
 require(msg.sender.call.value(amount)());

 balance[msg.sender] -= amount;
 return balance[msg.sender];
}
```

But is msg.sender an EOA or a contract ?

**Attacker Contract**



```
function reenterancyAttack() public payable {
    targetAddress.withdraw(amount)
}

function () public payable {
    if(address(targetAddress).balance >= amount) {
        targetAddress.withdraw(amount);
    }
}
```