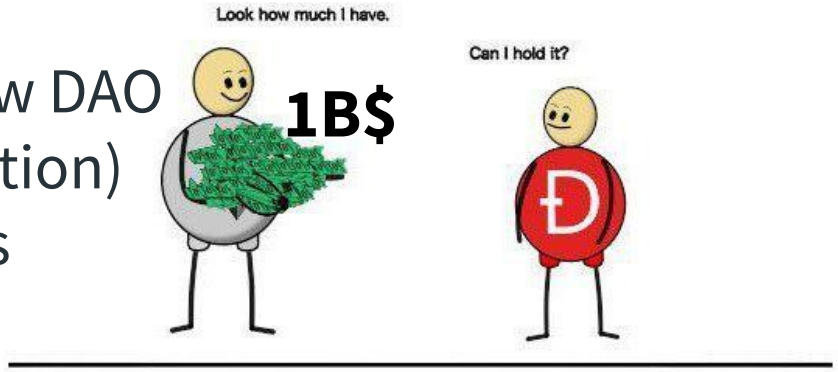
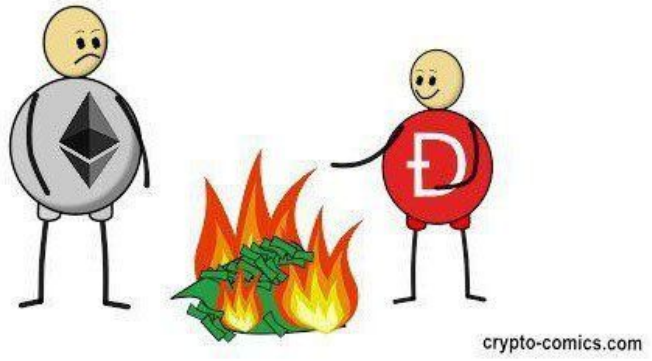
Technological & Economic risks associated to blockchain

Technological risks

The DAO hack

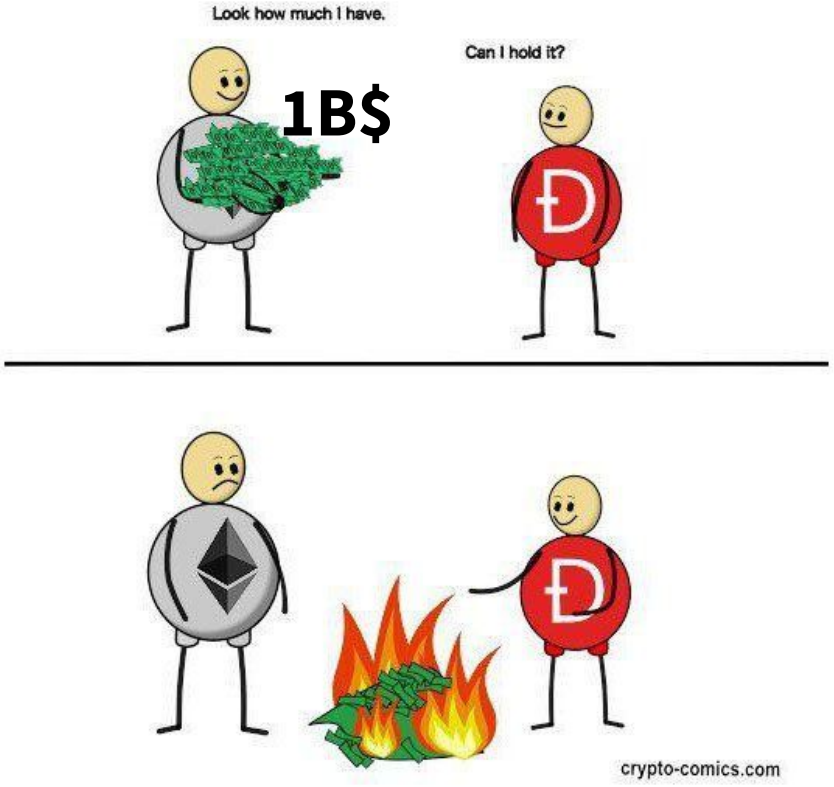
- Decentralized Investment Fund
  - Users pool money and vote on investments
  - Possibility to split the DAO in case of disagreement (avoid majority robbing the minority)
  - Splitting give your part of the funds to a new DAO
  - Problem
    - Funds are moved to the new DAO
    - User is called (can do an action)
    - User DAO tokens are burnts





The DAO hack

- Anonymous “hacker” use the exploit.
- Steal ⅓ of the funds.
- White hats take the remaining funds to give them back



Reentrancy (same function)

- Function makes an external call which calls back the same contract

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    require(msg.sender.call.value(amountToWithdraw)());
    // At this point, the caller's code is executed, and can call withdrawBalance again
    userBalances[msg.sender] = 0;
}
```

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    require(msg.sender.call.value(amountToWithdraw)());
    // The user's balance is already 0, so future invocations won't withdraw anything
}
```

Code is law?

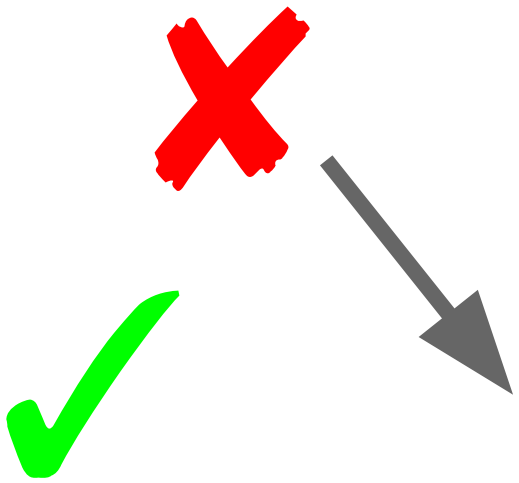
- Code is law
  - The contract specially stipulated that the code prevailed
  - Base layer should be agnostic
  - Legal risks for developers
- Early stage exception
  - Too much ETH
    - Staking
  - Should be user friendly
  - Avoid triggering regulators

Code is law?

- Code is law
  - The contract specially stipulated that the code prevailed
  - Base layer should be agnostic
  - Legal risks for developers
- Early stage exception
  - Too much ETH
    - Staking
  - Should be user friendly
  - Avoid triggering regulators

Code is law?

- Code is law
  - The contract specially stipulated that the code prevailed
  - Base layer should be agnostic
  - Legal risks for developers
- Early stage exception
  - Too much ETH
    - Staking
  - Should be user friendly
  - Avoid triggering regulators



ETHEREUM CLASSIC

Overflow

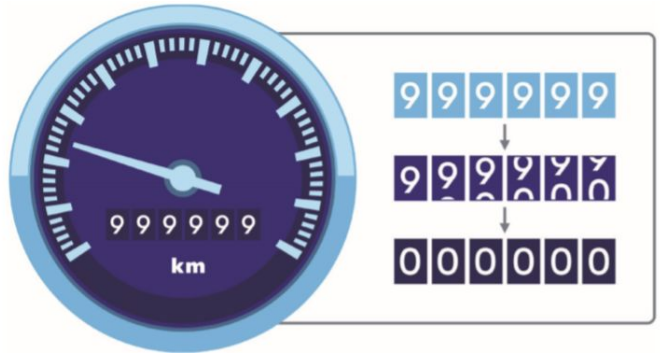
- Most common vulnerability
  - *, -, +

```
mapping (address => uint256) public balanceOf;

function transfer(address _to, uint256 _value) {
    /* Check if sender has balance */
    require(balanceOf[msg.sender] >= _value);
    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}

function transfer(address _to, uint256 _value) {
    /* Check if sender has balance and for overflows */
    require(balanceOf[msg.sender] >= _value && balanceOf[_to] + _value >= balanceOf[_to]);

    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}
```



Denial of service: Send/Transfer

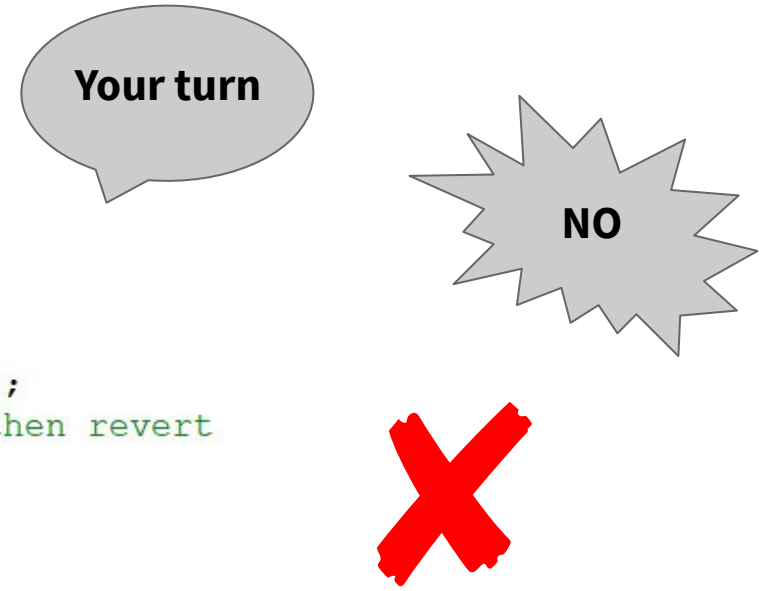
- Using transfer or send (while checking the return value) can allow the recipient to block the transaction.

```
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value > highestBid);

        require(currentLeader.send(highestBid));
        // Refund the old leader, if it fails then revert

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```



Storage Manipulation

- A large sized array allows to write anywhere in storage.
- Do not allow unbounded size arrays.

```
contract UnderflowManipulation {
    address public owner;
    uint256 public manipulateMe = 10;
    function UnderflowManipulation() {
        owner = msg.sender;
    }

    uint[] public bonusCodes;

    function pushBonusCode(uint code) {
        bonusCodes.push(code);
    }

    function popBonusCode()  {
        require(bonusCodes.length >=0);
        // this is a tautology
        bonusCodes.length--;
        // an underflow can be caused here
    }

    function modifyBonusCode(uint index, uint update)  {
        require(index < bonusCodes.length);
        bonusCodes[index] = update;
        // write to any index less than bonusCodes.length
    }
}
```

Gas limit attacks

- Functions can consume too much gas making them impossible to call


```
address[] private refundAddresses;
mapping (address => uint) public refunds;

function refundAll() public {
    for(uint x; x < refundAddresses.length; x++) {
    // arbitrary length iteration based on how many addresses participated
        refundAddresses[x].send(refunds[refundAddresses[x]]);
    }
}
```

Transaction ordering dependency (frontrunning)

- An actor can call a contract after a transaction has been broadcast but before it is mined.
- Use a commit - reveal pattern.

```
contract EthTxOrderDependenceMinimal {
    address public owner;
    bool public claimed;
    uint public reward;
    bytes32 public secret;

    function EthTxOrderDependenceMinimal() public {
        owner = msg.sender;
    }

    function setReward(bytes32 secret) public payable {
        require (!claimed);

        require(msg.sender == owner);
        owner.transfer(reward);
        secret = _secret;
        reward = msg.value;
    }

    function claimReward(bytes32 key) external {
        require (!claimed);
        require(keccak256(key) == secret);

        msg.sender.transfer(reward);
        claimed = true;
    }
}
```

Frontrunning

- Can look at a transaction before execution
- Execute it oneself if profitable

Signature Reuse

```
contract Sign {
...
    function doSomething(address _target, uint _amount bytes _data, _bytes _signature) public {
        require(verifieSignature(_target,_amount,_data,_signature));
        require(_target.call.value(_amount)(_data));
    }
}
```

```
contract Sign {
...
    mapping (bytes32 => bool) used;
    function doSomething(address _target, uint _amount bytes _data, uint _nonce, _bytes _signature) public {
        require(verifieSignature(_target,_amount,_data,_nonce,_signature));
        require(!used[keccak256(_signature)]);
        used[keccak256(_signature)=true;
        require(_target.call.value(_amount)(_data));
    }
}
```

```
contract Sign {
...
    mapping (bytes32 => bool) used;
    function doSomething(address _target, uint _amount bytes _data, uint _nonce, _bytes _signature) public {
        require(verifieSignature(_target,_amount,_data,_nonce,_signature));
        require(!used[keccak256(_target,_amount,_data,_nonce)]);
        used[keccak256(_target,_amount,_data,_nonce)]=true;
        require(_target.call.value(_amount)(_data));
    }
}
```

Reading the chain

- Everything is public on chain even “private” variables.
- Can sometimes solve it with a commit and reveal pattern

Reading the chain

```
contract RockPaperScissors {
...
function RPS(bytes32 _c1Hash, address _j2) payable {
        stake = msg.value; // La mise correspond à la quantité d'ethers envoyés.
        j1=msg.sender;
        j2=_j2;
        c1Hash=_c1Hash;
        lastAction=now;
}
    function play(Move _c2) payable {
        require(c2==Move.Null); // J2 has not played yet.
        require(msg.value==stake); // J2 has paid the stake.
        require(msg.sender==j2); // Only j2 can call this function.

        c2=_c2;
        lastAction=now;
    }

    function solve(Move _c1, uint256 _salt) {
        require(c2!=Move.Null); // J2 must have played.
        require(msg.sender==j1); // J1 can call this.
        require(keccak256(_c1,_salt)==c1Hash); // Verify the value is the commited one.

        if (win(_c1,c2))
            j1.send(2*stake);
        else if (win(c2,_c1))
            j2.send(2*stake);
        else {
            j1.send(stake);
            j2.send(stake);
        }
        stake=0;
    }
...
}
```

(t)RAB: (tests) - Review - Audit - Bounties

- Automated tests
  - Low cost. Good to identify what doesn't work but should (but not what works but shouldn't).
- Internal reviews
  - Low cost. Reviewers with good knowledge of interaction between systems.
- Audits
  - High cost. Auditors with high security skills. Allow to convince others that the contract is secure.
- Bounties
  - Intermediate cost. No bugs -> Free.

Every smart contract is its own bug bounty program...

-killerstorm

Auditor payout:

Audit_Price - Marketing_Expenses

Bounty hunter payout:

Nb_Vulnerabilities * Discovery_Rate * Bounty_Price

Vulnerability Overclassification

Vulnerability Overclassification

Minor - 1 ETH, 08/16/2018 6:41pm

Description

Due to imprecision in the shareEther computation, the result of the rounding may lead to ethers trapped in the contract.

For example, if 1 eth and 1 wei are sent, address A will receive 0.95 ether and address B will receive 0.05 ether, the remaining 1 wei will be trap in the contract.

1 ETH = 1E18 wei =
1000000000000000000 wei

Smart Contract Overengineering

- More code
  - More risks of making mistakes
  - More content for reviewers, auditors and bounty hunters to process

Write code only if

Risks created by additional security

<

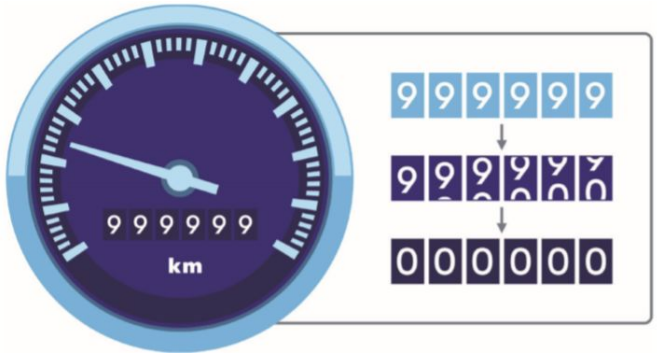
Extra protection offered by the code

Overengineering des smart contract

Overengineer des tests

Failure mode choice

- No free lunch!
- What is the “least bad” outcome
  - The function is blocked
    - SafeMath
      - 250+50= Revert
  - An extreme value is returned
    - CapMath
      - 250+50=255
  - Loop around
    - Default
      - 250+50=44



Failure mode choice

```
function finalize(uint _maxIt) public {
    uint endIt = startIt + _maxIt;
    uint currentIt = startIt;
    while (currentIt<endIt && currentIt<values.length) {
        gain = gain + values[currentIt];
        // Should not use SafeMath.
        reward[payers[currentIt]] =
            reward[payers[currentIt]] + values[currentIt] * bonus;
    }
    if (currentIt==values.length-1)
        owner.transfer(gain);
}
```

Protecting the user at the smart contract level?

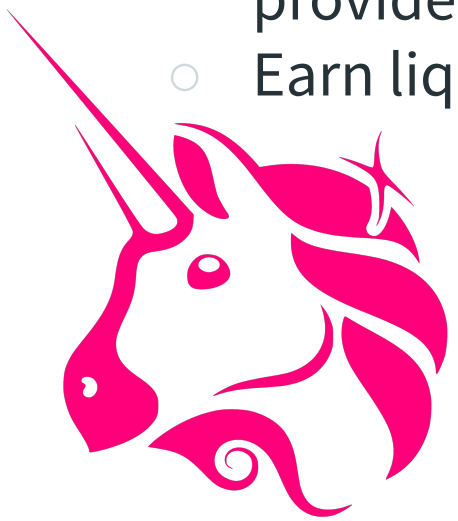
- Smart Contract
  - Protect from **malicious actions**.
- Graphical Interface
  - Protect from **stupid actions.**

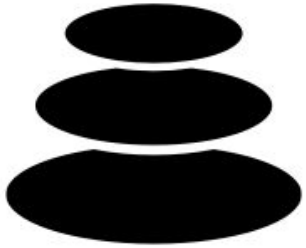
```
function transfer(address _to, uint256 _amount)
    public returns (bool success)
{
    require((_to != 0) && (_to != address(this)));
    .
    .
    .
```

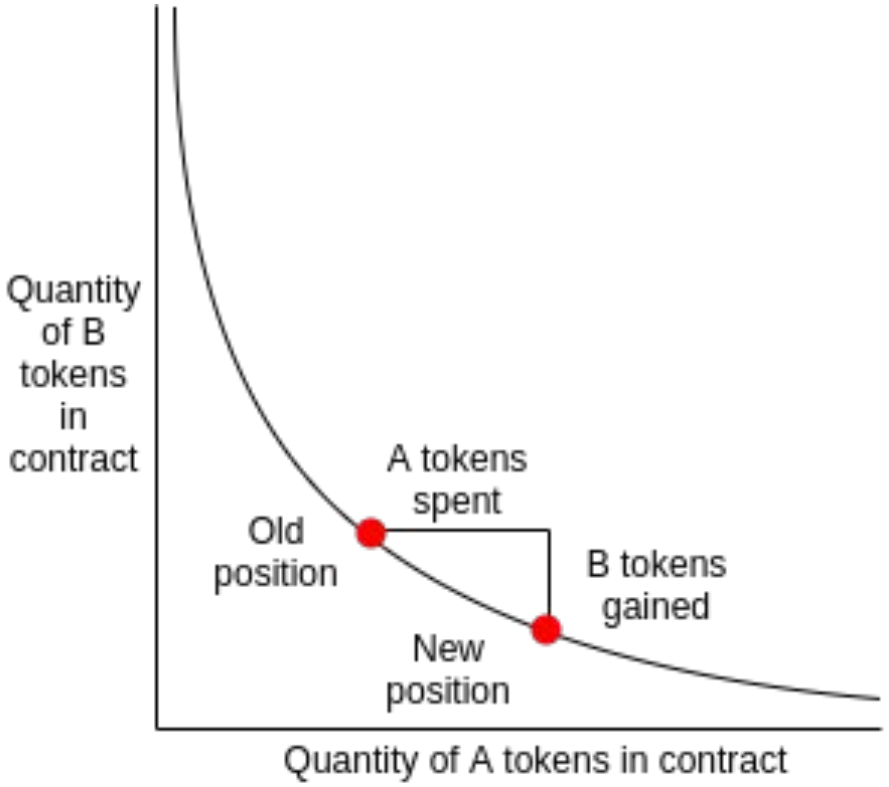
Economic risks

Automated market maker

- Automated market maker
  - Automatically buy/sell assets
  - Users can often deposits assets and act as liquidity providers
  - Earn liquidity fee (spread)

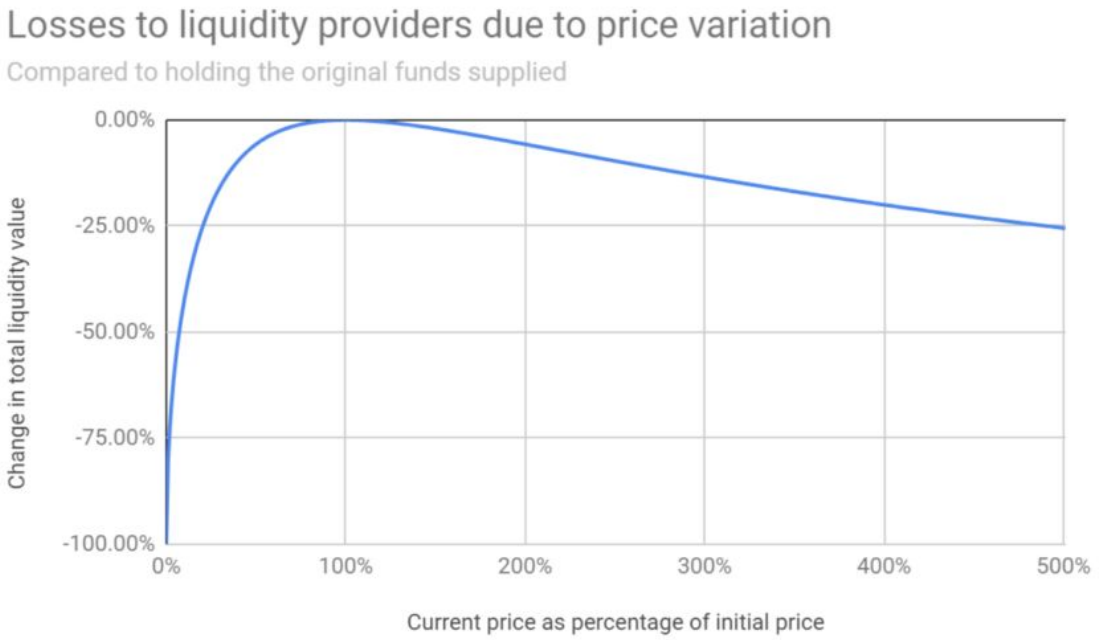






Impermanent loss

- Liquidity Providers
  - Win from fees
  - Lose from “Impermanent loss”
    - Get **more** of the **less** valuable asset



Weakest link

- Pool
  - 98% DAI (stable)
  - 2% cYFI (speculative)

Weakest link

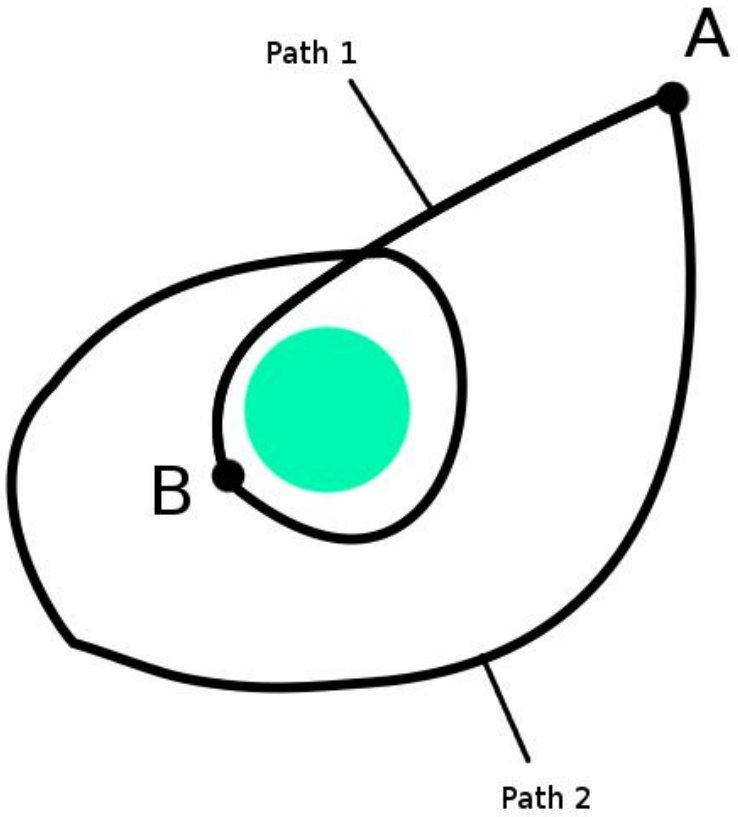
- Pool
  - 98% DAI (stable)
  - 2% cYFI (speculative)
- Safe?

Weakest link

- Pool
  - 98% DAI (stable)
  - 2% cYFI (speculative)
- Safe?
  - Can lose all the capital if cYFI drops to 0

Absence of path independence

- Automated market maker
  - Find ways to make it buy at a higher price than it sells



Absence of path independence

- Eminence hack
  a. Buy a large amount of EMN with DAI
  b. Burn a part to get eToken.
  c. Burn the other part back to the curve getting you DAI (at a higher price, since some EMN were burnt to get eToken so the supply is lower).
  d. Burn the other part back to the curve getting you DAI (at a higher price, since some EMN were burnt to get eToken so the supply is lower).
  e. Burn the eToken to get EMN back.
  f. Burn EMN to get DAI back.

Absence of path independence

- Eminence hack
  - 15 m$ loss
  - Attacker gave back half of them

Flash loans and atomicity

- Flash loan
  - 0s loan
  - Atomic
  - Cost only paid at the end
  - No collateral
- Use
  - Arbitrage opportunities
  - No capital requirements
  - No execution risk
  - No counterparty risk
  - Worse case trade reverts and lose only gas fee (few $)

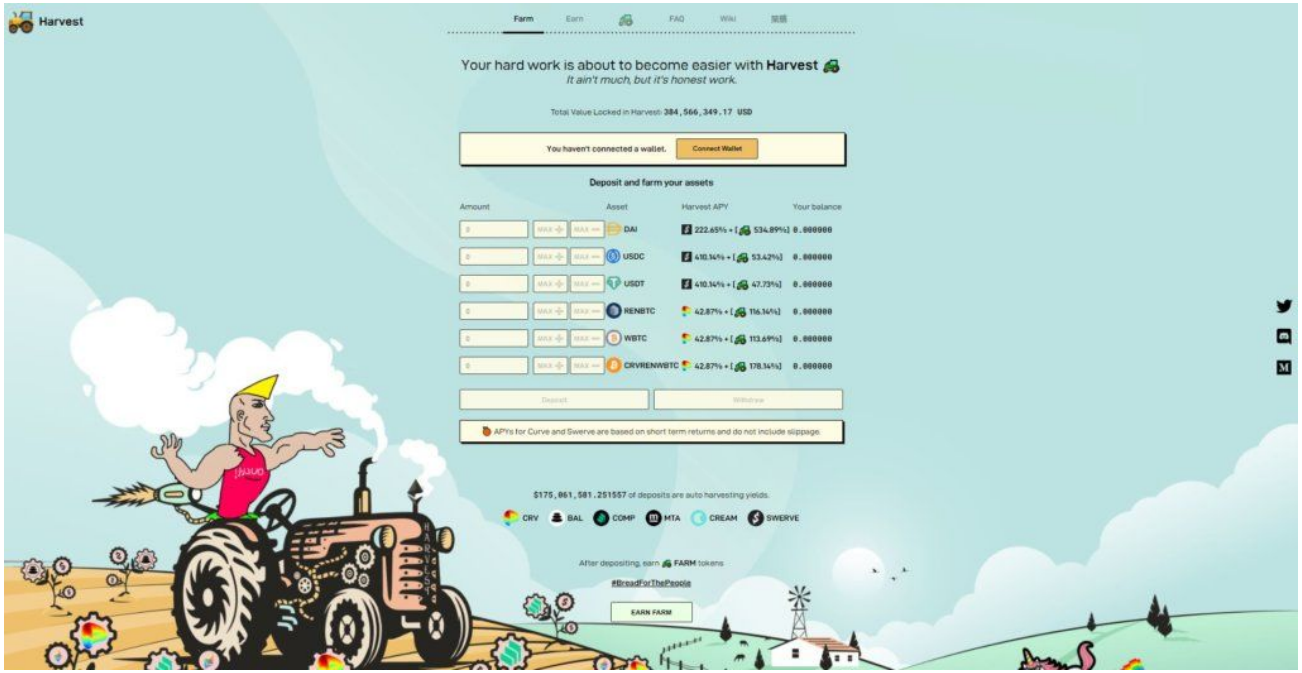
Here is 1B$,
no collateral needed but
you must reimburse in 0s.

Flash loans and atomicity

- Issue
  - Take huge loan
  - Buy the asset
  - Deposit the asset (high credit cause asset price high)
  - Sell asset
  - Withdraw asset (lower debit cause asset price normal)
  - Repeat with a few % benefits each cycle

Flash loans and atomicity

- Harvest finance
  - Can deposit capital
  - Capital is used in another protocol
  - Yield automatically sold



Governance attack

- Maker
  - Token holders delegate to “hats”
  - Hat with the highest amount of tokens can perform actions
  - Incentive to delegate to “good” hats (skin in the game)

Governance attack

- Maker
  - Token holders delegate to “hats”
  - Hat with the highest amount of tokens can perform actions
  - Incentive to delegate to “good” hats (skin in the game)
- Flashloan
  - Borrow tons of MKR
  - Delegate to a “bad hat”
  - Execute action
  - Reimburse loan
  - No skin in the game

Governance attack

- Attack
  - Known team
  - Only whitelist an address for an Oracle
  - Inform the team of the issue
  - Hot topic right now

Collateralized Debt Position

- Deposit assets
  - Earn interest
  - Borrow other assets
- Borrow assets
  - Pay interest
  - Short
  - Leverage
  - Use
- Liquidation
  - Ratio borrowed/deposited too low
  - Deposits auctioned

Collateralized Debt Position

- Market risk
  - Deposit prices drop too fast
  - Ratio < 1
  - System debt
    - Reimbursed by token holders

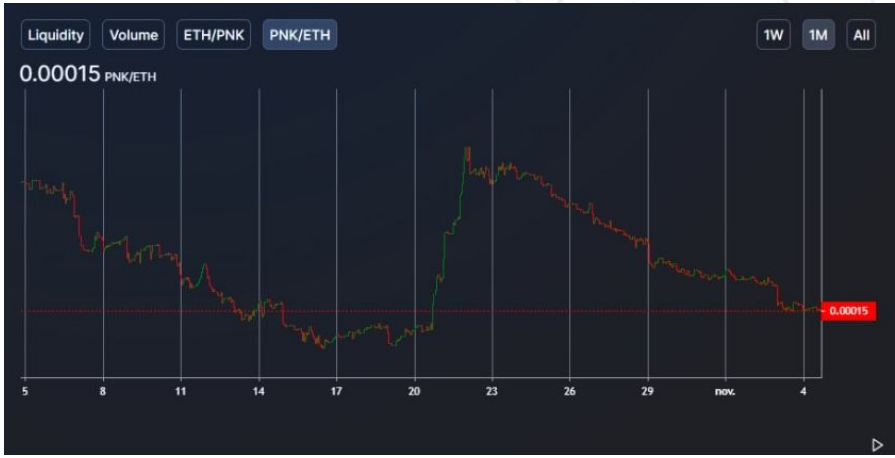
Collateralized Debt Position

- Counterparty risk
  - Entities governing assets become malicious
- Smart contract risk
  - Smart contract of assets broken
- Effect
  - Deposit worthless
    - Reimbursed by token holders
  - Infinite minting
    - Attacker can borrow everything with worthless assets
    - May have limits (maker)

Collateralized Debt Position

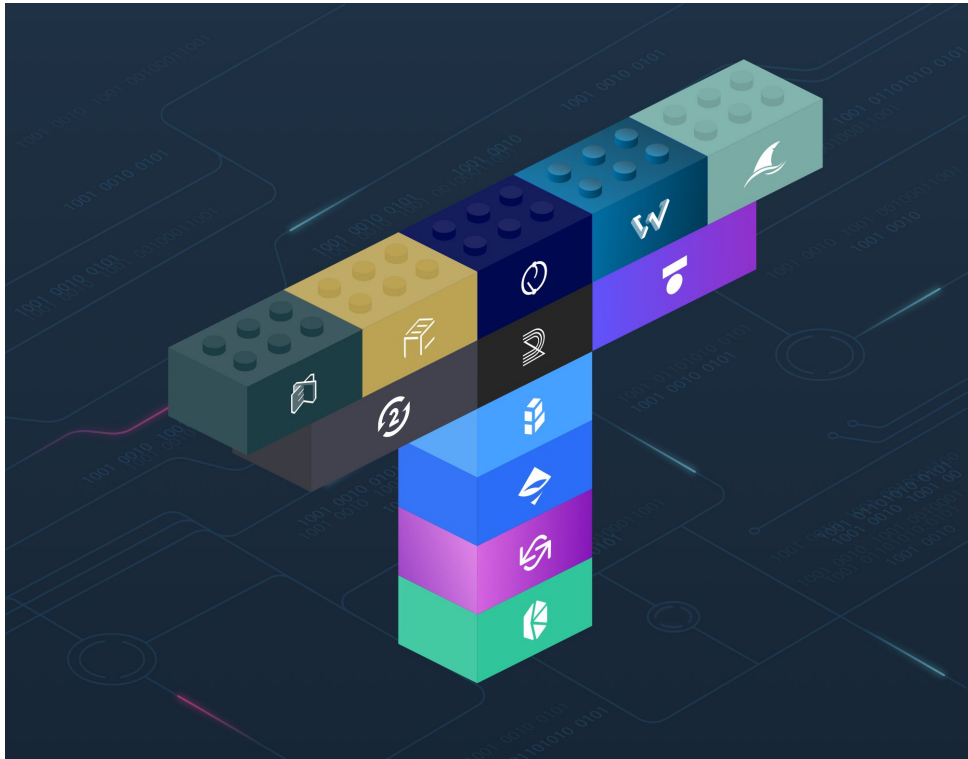
- Liquidation risk
  - No one buy the sold asset
  - Assets sold at 0 (Maker)
- Oracle risk
  - Real time oracle
  - Oracle malfunction
  - Price manipulation

Washtrading exchanges



Defi Lego

- Composability / Open execution
  - Everyone can build on top of other protocols
  - Permissionless
- Risks
  - Complexity
  - Unplanned interactions
  - Multiple points of failure



Often better than alternatives

- Exchanges hack in 2019
  - Centralized exchanges: ~33m$
  - Decentralized Exchanges: negligible
  - Similar volume
- Way more transparency
  - Centralized exchanges hacked continuously over years

Recoveries by forking

- Ethereum: DAO hack
  - Training wheels
  - Unlikely to happen again
- Steem / Hive
  - Large entities owning enough tokens to control governance
  - Used maliciously
  - Community fork creating Hive

Questions

- Code is law?
  - Yes
  - No
  - Only on chains with this philosophy
- “Economic hack”
  - Arbitrage?
  - Price manipulation?
  - Bidding 0 on auctions?
- Malicious governance?
- Forking to remove “bad” actors?

Thanks!

I am Clément Lesaege

Working on a decentralized court project :

Kleros.io.

You can find me at:

clement@lesaege.com