CSBC2000

Week 2 | Class 4

Smart Contract Security

Last Class

- Learnt various cryptography techniques
- Looked at quantum resistance
- Saw how these concepts are used in blockchain
- (almost) saw secure multiparty computation using Enigma

This Class

- Quick JS Blockchain update
- Moonbeam demo revisited
- An overview of blockchain security and major incidents
- Live walkthrough of DAO exploit

Financial Attacks: Mixers and Tumblers

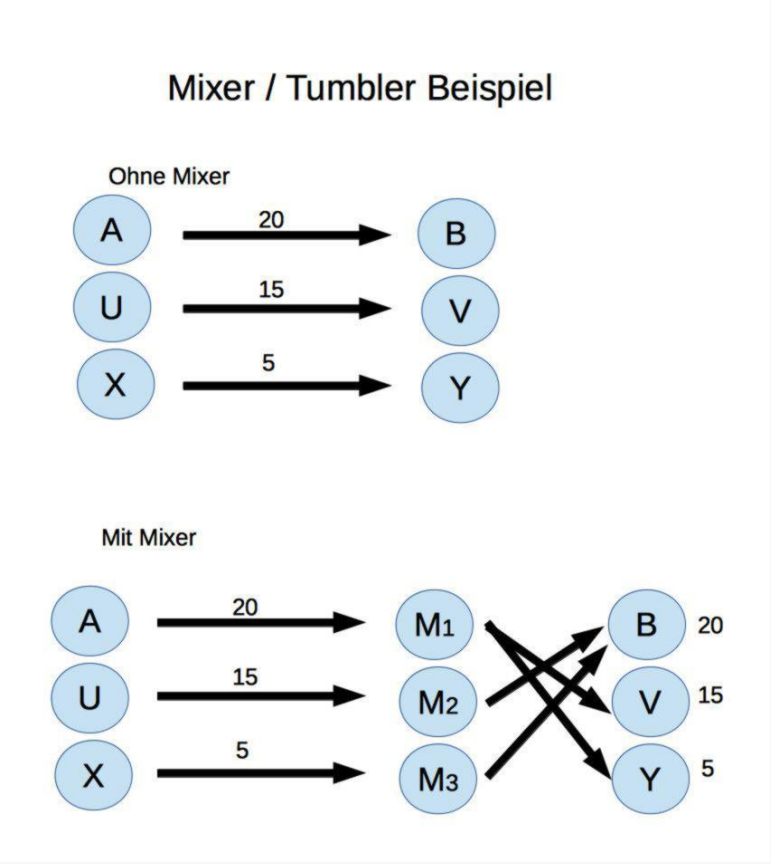
- Say Evil Steve successfully exploited a smart contract and steals funds
- Since everyone can see the transactions publicly, they see his address there and collectively choose to stop trading with him
- But accounts are cheap to creat in Ethereum; just need to generate an address (keypair) and announce it to the blockchain during a tx

Financial Attacks: Mixers and Tumblers

- This allows Evil Steve to generate several addresses that he owns and make legitimate transactions on them
- Once he has accumulated value on those, he can implement a *mixer/tumbler* to get the value into circulation from the malicious address
- Through a course of several transactions, the address can make anonymized transactions by mixing them with those of the legitimate accounts

Financial Attacks: Mixers and Tumblers

- Txs are made from the malicious account to intermediate "mixer" addresses which keep track of all the inputs and randomize outputs
- Used widely in illegal Bitcoin transactions (Deepweb)
- These are extremely hard to perform *cryptoforensics* on



Mempool Attacks: Flooding

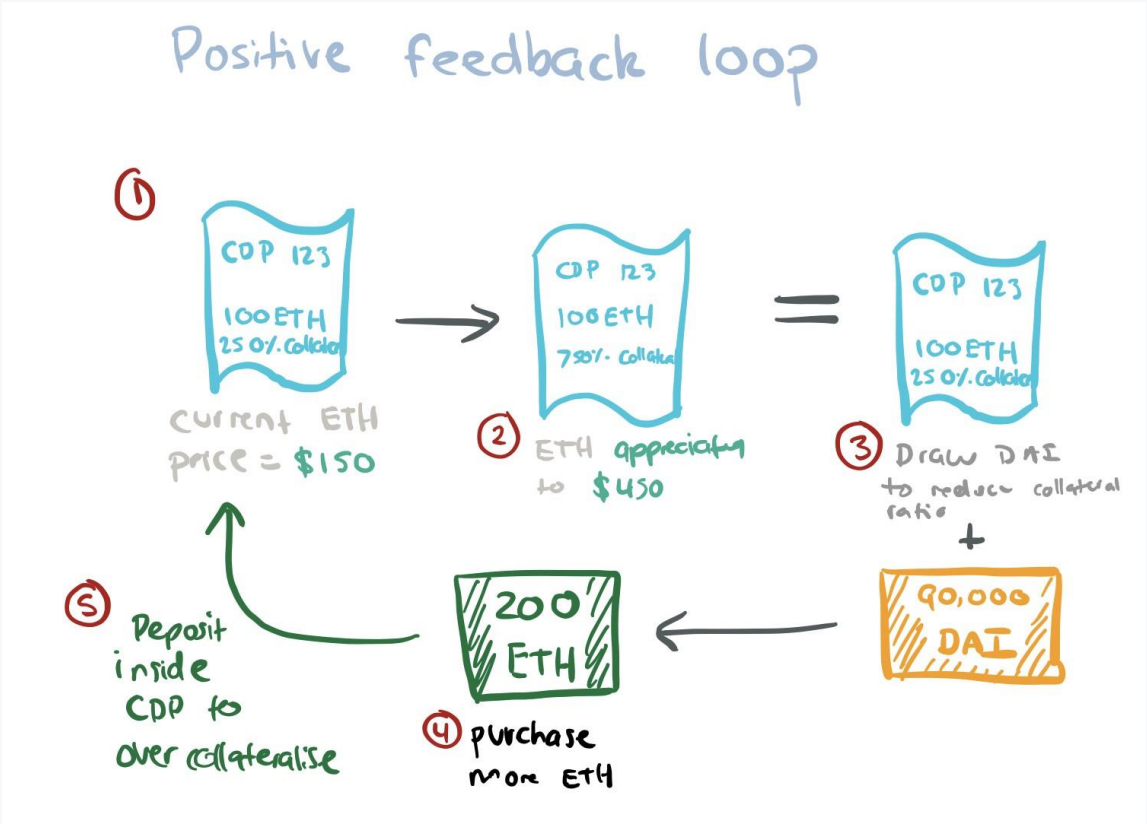
- Mempools are where transactions go while they are waiting to be mined into a block

- Each peer has its own mempool and if it successfully mines a block, it can put transactions into it (based on miner fees) then broadcast it to the network

Mempool Attacks: Flooding

- This can be exploited in certain scenarios
- MakerDAO: users put ETH or other crypto-assets up as collateral on the Maker platform to then withdraw a portion of the value of those assets in the form of brand-new DAI
- Works based on a "positive feedback loop"

Mempool Attacks: Flooding



Mempool Attacks: Flooding

- On March 12, 2020, there were an unusually high number of very low gas fee txs
- These were definitely not getting mined, but they prevented other txs from going through
- “The bots hammered the mempool with transactions that were never intended to be finalized. These ‘Hammerbots’ consumed mempool resources by issuing extremely high rates of replacement transactions without any corresponding increase in gas”

Mempool Attacks: Flooding

- Further, there are bots on Maker known as "Keepers" that maintain collaterallization in the network
- These bots couldn't get their transactions in to maintain the value of DAI because they did not track the order of pending transactions
  - They could "probabilistically" get transactions in though
- The attackers noticed this bug, and submitted several strong gas fee liquidation transactions with 0 collateral at a rate faster than keepers
- Enough of these went through that the attackers walked away with $8M, with 0 consequences

Mempool Attacks: Front-running

- In a trading floor, there's typically a lot of chaos (as we might know from Wolf of Wall Street)
- In the chaos, people might overhear good intel from a party and then try to make a transaction to exploit it before the original party gets to make it
- This is termed as "front-running"
- Since mempool txs are publicly accessible, the info in them can be exploited
- E.g. Evil Steve can see Alice's stock bid in a decentralized exchange mempool and make that same transaction at a higher miner fee
  - And maybe broadcast it to more nodes than Alice
- This will let him make the bid before her

Mempool Attacks: Displacement Front-running

- In the first type of attack, it is not important for Alice's function call to run after Evil Steve runs his function
- Alice's can be orphaned or run with no meaningful effect
- Examples of displacement include
  - Alice trying to register a domain name and Mallory registering it first
  - Alice trying to submit a bug to receive a bounty and Mallory stealing it and submitting it first
  - Alice trying to submit a bid in an auction and Mallory copying it.
- This attack is commonly performed by increasing the gasPrice higher than network average, often by a multiplier of 10 or more.

Mempool Attacks: Insertion Front-running

- For this type of attack, it is important to the adversary that the original function call runs after their transaction
- In an insertion attack, after Evil Steve runs his function, the state of the contract is changed, and he needs Alice’s original function to run on this modified state
- For example, if Alice places a purchase order on a blockchain asset at a higher price than the best offer, Evil Steve will insert two transactions: he will purchase at the best offer price and then offer the same asset for sale at Alice’s slightly higher purchase price
- If Alice’s transaction is then run after, Evil Steve will profit on the price difference without having to hold the asset.
- As with displacement attacks, this is usually done by outbidding Alice's transction in the gas price auction.

Mempool Attacks: Suppression Front-running

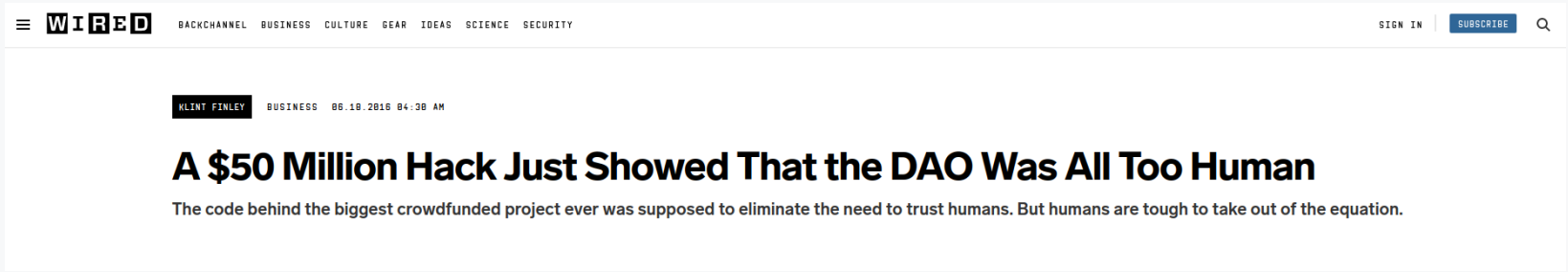
- A.k.a Block Stuffing attacks
- After Evil Steve runs his function, he tries to delay Alice from running her function
- "The attacker sends multiple transactions with a high gasPrice and gasLimit to custom smart contracts that assert (or use other means) to consume all the gas and fill up the block's gasLimit"

Smart Contracts in Ethereum

- First are deployed to the blockchain, which occurs when it is mined into a block
- This block will be the "genesis block" of the smart contract
- Can be funded with tokens since they have an Eth address
- Can define what to do with the money sent into it based on code (Solidity logic)

Smart Contracts in Ethereum

- Thus, Solidity devs must be careful!
- A vulnerability can be quite devastating



Smart Contract Bugs: Re-entrancy

- Imagine you have an escrow smart contract
- You have a buyer and a seller; buyer puts money in escrow when real world asset transfer occurs
- And many buyers/sellers can do this by invoking smart contract functions
- Since smart contracts are also Eth addresses, they can invoke functions in other smart contracts
- Note: if currency is sent to a smart contract, it can invoke "default" code

Smart Contract Bugs: Re-entrancy

- Here's some code that can perform escrow

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

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
    require(success);
    userBalances[msg.sender] = 0;
}
```

Smart Contract Bugs: Re-entrancy

- Since smart contracts can have default functions, Line 6 can actually invoke malicious code in the calling smart contract!

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

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
    require(success);
    userBalances[msg.sender] = 0;
}
```

Smart Contract Bugs: Re-entrancy

- An example (we'll see this in action later)

```
function () {
  DefaultFunc(msg.sender,msg.value,a,dumbDAO(daoAddress).balances(this)-1);
  while (a<5) {
      a++;
      arr.push(a); //to help debug
      //  if (daoAddress.balance-2*msg.value < 0){
      if (a==4){
          dumbDAO(daoAddress).transferTokens(transferAddress,dumbDAO(daoAddress).balances(this)-1);
      }
      dumbDAO(daoAddress).withdraw(this);
  }
}
```

Smart Contract Bugs: Re-entrancy

- This function can trigger multiple calls to the same block of withdraw code, which steals money from the victim contract!
- This is known as *re-entrancy* since the same function is being entered simultaneously

```
function () {
  DefaultFunc(msg.sender,msg.value,a,dumbDAO(daoAddress).balances(this)-1);
  while (a<5) {
      a++;
      arr.push(a); //to help debug
      //  if (daoAddress.balance-2*msg.value < 0){
      if (a==4){
          dumbDAO(daoAddress).transferTokens(transferAddress,dumbDAO(daoAddress).balances(this)-1);
      }
      dumbDAO(daoAddress).withdraw(this);
  }
}
```

Smart Contract Bugs: Re-entrancy

- Best way to protect is to make sure to finish all state modifications before potentially invoking external code

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

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    (bool success, ) = msg.sender.call.value(amountToWithdraw)("");
    require(success);
}
```

Smart Contract Bugs: Re-entrancy

- Also avoid calling functions that call external functions before state is fully modified

```
// INSECURE
mapping (address => uint) private userBalances;
mapping (address => bool) private claimedBonus;
mapping (address => uint) private rewardsForA;

function withdrawReward(address recipient) public {
    uint amountToWithdraw = rewardsForA[recipient];
    rewardsForA[recipient] = 0;
    (bool success, ) = recipient.call.value(amountToWithdraw)("");
    require(success);
}

function getFirstWithdrawalBonus(address recipient) public {
    require(!claimedBonus[recipient]); // Each recipient should only be able to cl

    rewardsForA[recipient] += 100;
    withdrawReward(recipient); // At this point, the caller will be able to execut
    claimedBonus[recipient] = true;
}
```

Questions/Comments?

Let's code!