D7: Front-running

Race conditions

Portland State
Computer Science

#7: Front-running

- A form of a race condition **time-of-check vs time-of-use (TOCTOU) race conditions** and **transaction ordering dependence (TOD)**
  - A classic problem in operating systems
  - 15.8% of all smart contracts contain a transaction ordering dependence vulnerability
  - Allows a miner to subvert a pending transaction before it has been committed onto the ledger.
- Term "front-running" from financial trading

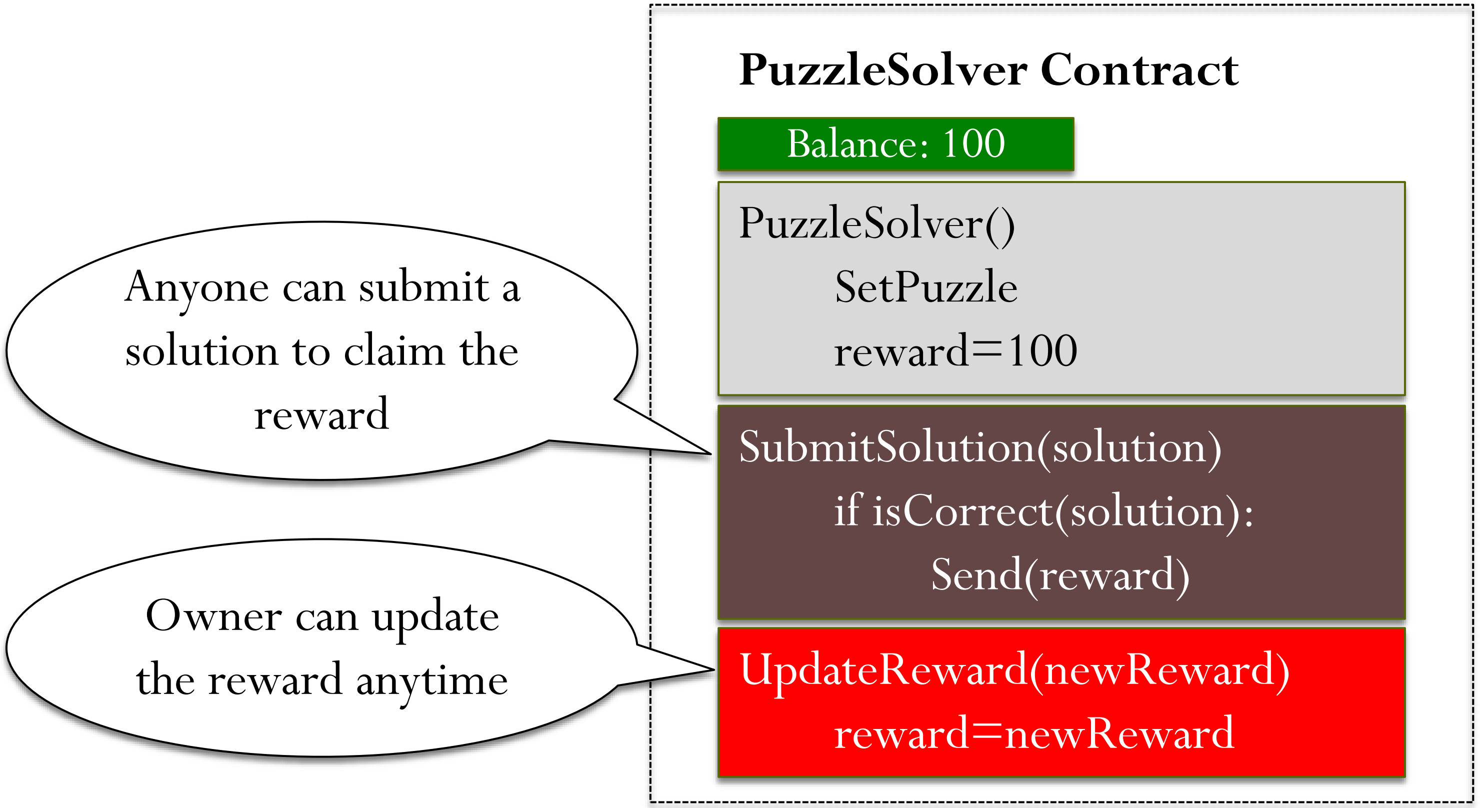
Front-running in stock trading

- Trading originally done on stock market floor by paper
  - Orders carried by hand between traders
  - Broker receives a buy order from client
  - Places his/her own order for themselves in front to clear lower-priced sell orders
  - Stock price increases and broker sells at higher price at the expense of client
  - Practice is outlawed for brokers in real-life, but such laws don't apply on blockchain
- TOC
  - Client checking price and deciding to buy
- TOU
  - Getting a different price upon execution due to trader front-running

Walkthrough scenario #1

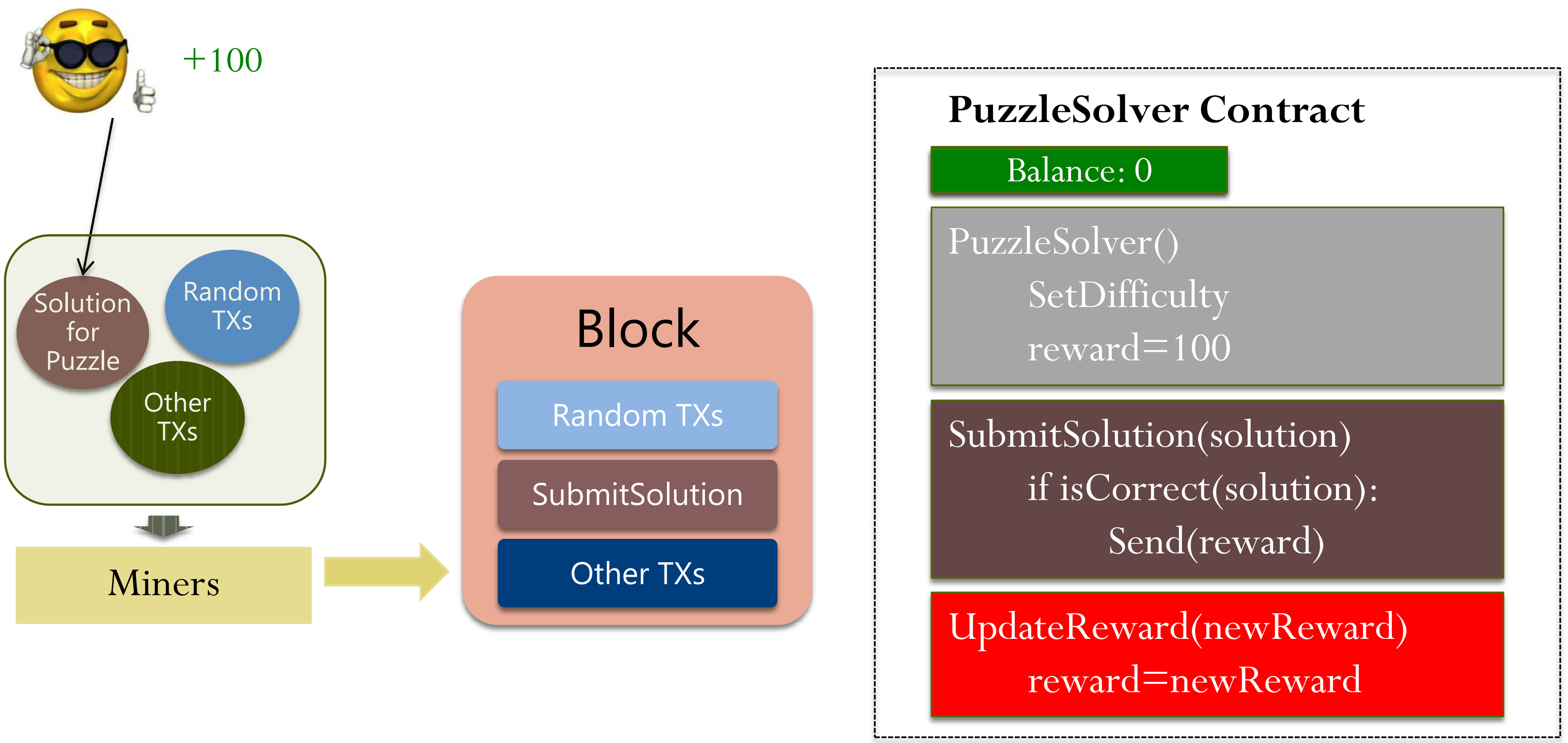
- A prime factoring smart contract publishes an RSA number
  `N = prime1 x prime2`
- A call to its **submitSolution()** public function with the values for `prime1` and `prime2` rewards the caller.
- Alice successfully factors the RSA number and submits a solution.
- Attacker on the network sees Alice's transaction (containing the solution) waiting to be mined and resubmits it as his/her own with a higher gas price
- Attacker's transaction gets picked up first by miners due to the higher paid fee
- The attacker wins the prize.

Walkthrough scenario #2


PuzzleSolver Contract
Balance: 100
Anyone can submit a solution to claim the reward
Owner can update the reward anytime

```
PuzzleSolver()
        SetPuzzle
        reward=100
SubmitSolution(solution)
        if isCorrect(solution):
                Send(reward)
UpdateReward(newReward)
        reward=newReward
```

- Expected operation

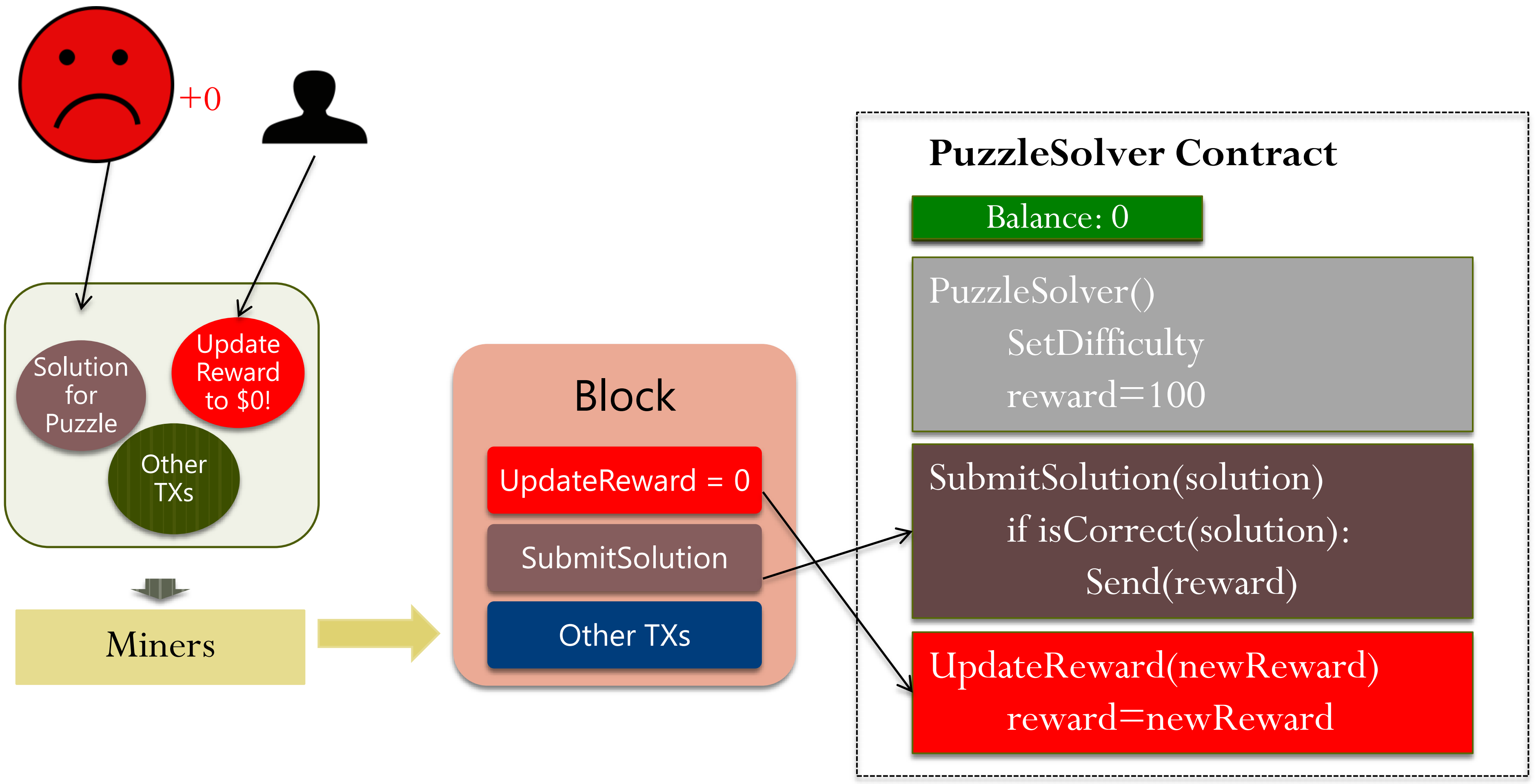


```
PuzzleSolver()
    SetDifficulty
    reward=100

SubmitSolution(solution)
    if isCorrect(solution):
        Send(reward)

UpdateReward(newReward)
    reward=newReward
```

- Malicious contract operator scenario

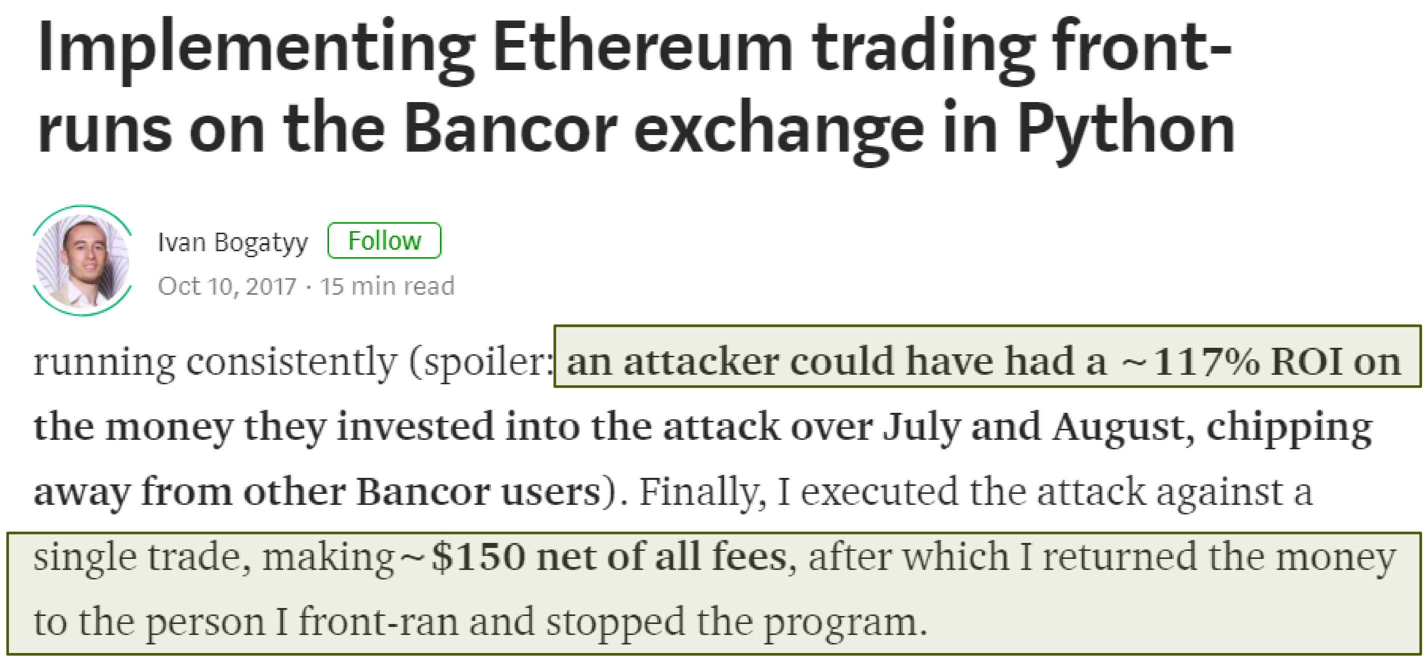


Intuition

- Observed state != execution state
  - Transactions do not have atomicity property
- Can be coincidence
  - Two transactions happen at the same time
- But, can be malicious

Example

- Front-running the Bancor market-maker for ERC-20 tokens
  - Matches buyers and sellers of tokens automatically within a smart-contract



Implementing Ethereum trading front-runs on the Bancor exchange in Python

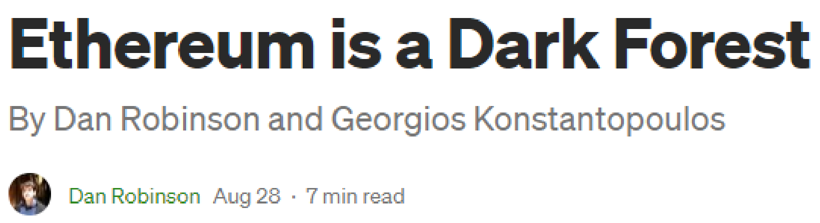
Ivan Bogatyy Follow
Oct 10, 2017 · 15 min read

running consistently (spoiler: **an attacker could have had a ~117% ROI on the money they invested into the attack over July and August, chipping away from other Bancor users**). Finally, I executed the attack against a single trade, making ~**$150 net of all fees**, after which I returned the money to the person I front-ran and stopped the program.

Mechanism

- Buyer submits a transaction to purchase tokens to the network
- Broadcast immediately to other nodes as a pending transaction and added to common queue
  - Not confirmed until the block confirmation hash mined (~20 seconds)
  - Order of pending transactions is malleable until then
- Miners sort transactions by gas price willing to be paid
- Any user running a full-node can spot a pending transaction and insert their own transaction in front of it by paying 1 Wei more per gas.
- If a large BUY is about to happen, BNT price will increase (following deterministic formula in contract)
  - Put buy in before that transaction to get an instant appreciation of your tokens and a guaranteed return on your investment
- If a large SELL is about to happen, BNT price will decrease
  - Put a sell in before to get the higher price for you tokens
  - Link (6 min)

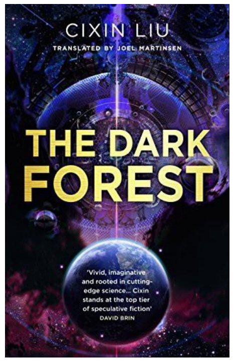
Example: Rescuing funds from contracts



Ethereum is a Dark Forest

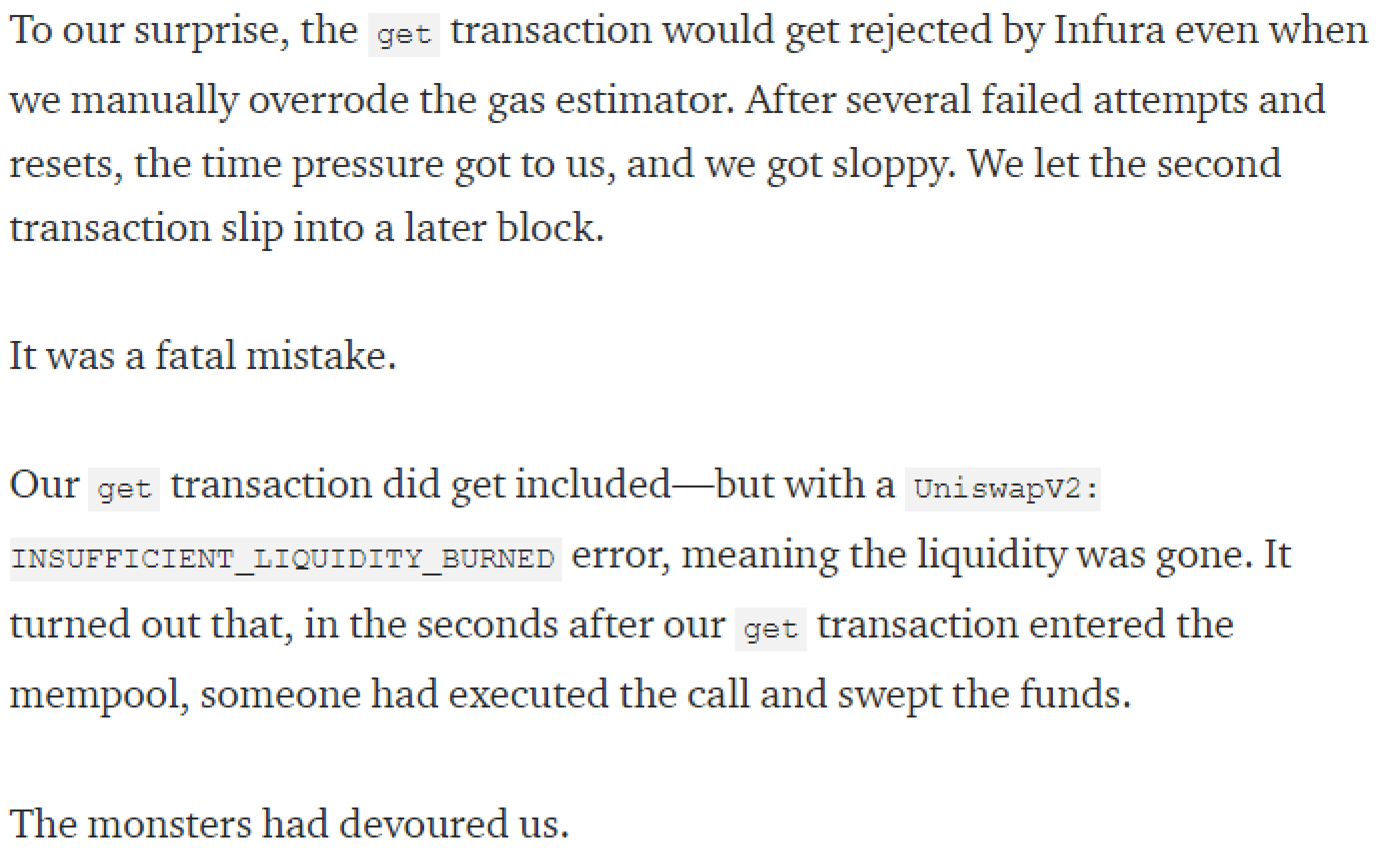
By Dan Robinson and Georgios Konstantopoulos

Dan Robinson Aug 28 · 7 min read


CIXIN LIU
TRANSLATED BY JOEL MARTINSEN
THE DARK FOREST
'Vivid, imaginative and rooted in cutting-edge science... Cixin stands at the top tier of speculative fiction' DAVID BRIN

- 2nd book in "Three-Body Problem" trilogy
  - Survival of lower civilizations depends upon not being discovered by higher ones
- Apex predators tracking Ethereum mempool
  - Arbitrage bots monitor pending transactions and attempt to exploit profitable opportunities created by them
  - *"If the chain itself is a battleground, the mempool is something worse: a dark forest..Detection means certain death at the hands of advanced predators"*
- Rescues of vulnerable smart contracts require obfuscation to hide from Arbitrage bots
  - *"If I submitted a transaction calling burn, it would be like a flashing "free money" sign pointing directly at this profitable opportunity. If these monsters were really in the mempool, they would see, copy, mutate, and front-run my transaction, taking the money before my transaction was included."*

- Predator wins (taking $12,000 in ETH)



To our surprise, the `get` transaction would get rejected by Infura even when we manually overrode the gas estimator. After several failed attempts and resets, the time pressure got to us, and we got sloppy. We let the second transaction slip into a later block.

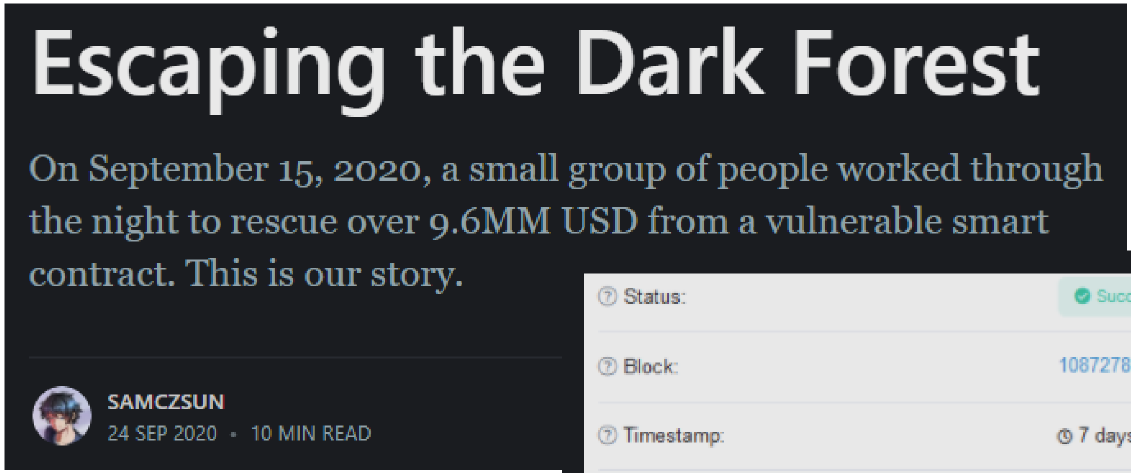
It was a fatal mistake.

Our `get` transaction did get included—but with a `UniswapV2: INSUFFICIENT_LIQUIDITY_BURNED` error, meaning the liquidity was gone. It turned out that, in the seconds after our `get` transaction entered the mempool, someone had executed the call and swept the funds.

The monsters had devoured us.

- But subsequently…

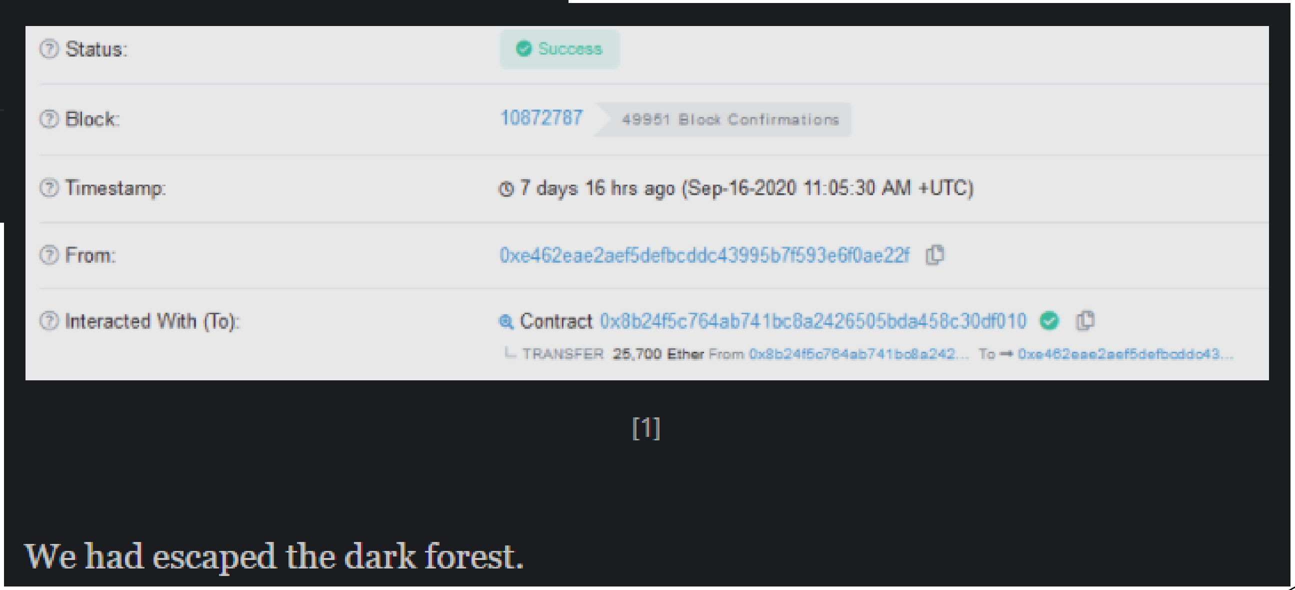
- Similar vulnerability putting ~$10M USD at risk (9/2020)
- Combine obfuscation from previous rescue attempt with cooperation
  - Convince Chinese SharkPool miners to include a transaction in a block that they would attempt to mine without broadcasting it to others
    - *If ever there was a time to appeal to a miner to include a transaction without giving front-runners the chance to steal it, it was now.*
    - "WhiteHat" API built on the spot once translation issues overcome



Escaping the Dark Forest

On September 15, 2020, a small group of people worked through the night to rescue over 9.6MM USD from a vulnerable smart contract. This is our story.

SAMCZSUN
24 SEP 2020 • 10 MIN READ



Status:	Success
Block:	10872787 49951 Block Confirmations
Timestamp:	7 days 16 hrs ago (Sep-16-2020 11:05:30 AM +UTC)
From:	0xe462eae2aef5defbcddc43995b7f593e6f0ae22f
Interacted With (To):	Contract 0x8b24f5c764ab741bc8a2426505bda458c30df010 └ TRANSFER 25,700 Ether From 0x8b24f5c764ab741bc8a242... To → 0xe462eae2aef5defbcddc43...

[1]

We had escaped the dark forest.

Code vulnerability example

- Can also be leveraged by a malicious client
- Bank contract
  - `userBalances` mapping to track account balances per user address (in storage that only changes after block committed)
  - `transfer()` moves balance from one user to another if sufficient funds
  - `withdrawBalance()` zeros out account and sends user remaining balance
  - Issue?

```
contract myBank {
    mapping (address => uint) private userBalances;

    function transfer(address to, uint amount) {
        if (userBalances[msg.sender] >= amount) {
            userBalances[to] += amount;
            userBalances[msg.sender] -= amount;
        }
    }
    function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];
        require(msg.sender.send(amountToWithdraw)());
        userBalances[msg.sender] = 0;
    }
}
```

- Cross-function race condition with external calls
  - Found in The DAO (along with re-entrancy)
  - Simultaneous execution of `transfer()` and `withdrawBalance()`

```
contract myBank {
    mapping (address => uint) private userBalances;

    function transfer(address to, uint amount) {
        if (userBalances[msg.sender] >= amount) {
            userBalances[to] += amount;
            userBalances[msg.sender] -= amount;
        }
    }
    function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];
        require(msg.sender.send(amountToWithdraw)());
        userBalances[msg.sender] = 0;
    }
}
```

  - What would you do to avoid this?

Remediation

- Mutexes, semaphores/locks, condition variables, etc. (critical sections) when external calls unavoidable
  - But, prone to deadlock and livelock issues.

```
contract mutexExample {
   mapping (address => uint) private balances;
   bool private lockBalances;
   function deposit() payable public {
      require(!lockBalances);
      lockBalances = true;
      balances[msg.sender] += msg.value;
      lockBalances = false;
   }

   function withdraw(uint amount) payable public {
      require(!lockBalances && amount > 0 && balances[msg.sender] >= amount);
      lockBalances = true;
      if (msg.sender.call(amount)()) {
         balances[msg.sender] -= amount;
      }
      lockBalances = false;
    }
}
```

D8: Time Manipulation

Portland State
Computer Science

#8: Time manipulation

- also known as **timestamp dependence**
- Time tracked via `block.timestamp` (or its Solidity alias `now`)
  - Locking a token sale
  - Unlocking funds at a specific time for a game
- Timestamp value determined by miner that successfully mines block
  - Miner has leeway to manipulate actual value
- Contracts must avoid relying strongly on advertised time
  - e.g. using it to generate random numbers critical to smart contract execution

Example #1

- Lottery that uses block.timestamp to generate numbers
- Miner either
  - Selects block.timestamp so he/she can win
  - Otherwise, selects block.timestamp so no one else can win in current block

Code vulnerability example #1

- A **game** pays out the very first player at midnight.

```
function play() public {
  require(now > 1521763200 && neverPlayed == true);
  neverPlayed = false;
  msg.sender.transfer(1500 ether);
}
```

- A malicious **miner** includes his or her attempt to win the game and sets the timestamp to midnight.
  - Just before midnight, miner submits attempt and begins mining the block
  - Even, though real current time slightly before midnight, miner includes timestamp that is "close enough" to be accepted by all nodes in network

D10: Everything else (Unknown unknowns)

Renamed since unknown unknowns would be blank

Portland State
Computer Science

Usage and logic errors

Logic errors

```
def add_player():
        if not self.storage["player1"] and msg.value > 1000:
                self.storage["player1"] = msg.sender
                self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
                return(1)
        elif not self.storage["player2"] and msg.value > 1000:
                self.storage["player2"] = msg.sender
                self.storage["WINNINGS"] = self.storage["WINNINGS"] + msg.value
                return(2)
        else:
                return(0)
```

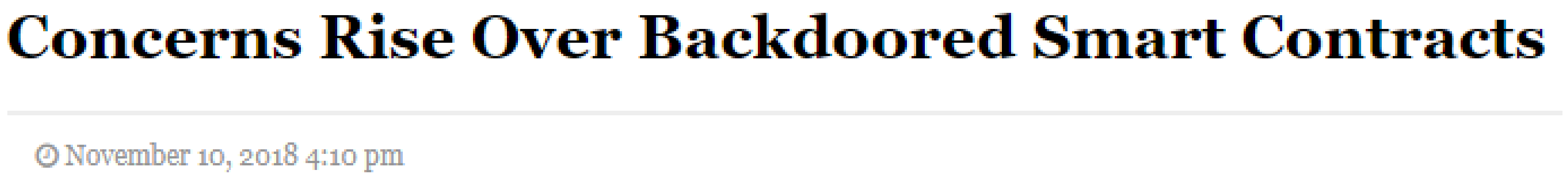
- Code takes your money if you send less than 1000
- Code takes your money if you are not `player1` or `player2`

No penalty for bad behavior

- Game implements bit-commitment protocol
  - 2-players publish a keyed hash of their random numbers
  - Subsequently reveal the numbers to determine winner
- Upon seeing a key that reveals a committed number, the other player fails to reveal his/her key if it is a losing move
  - e.g. `player1` opens move, but `player2` refuses to open move since there is do no incentive to do so
  - Must add a deposit to play and timeout player (forfeiting the deposit)

Malicious contracts

Intentional Backdoors



Concerns Rise Over Backdoored Smart Contracts

November 10, 2018 4:10 pm

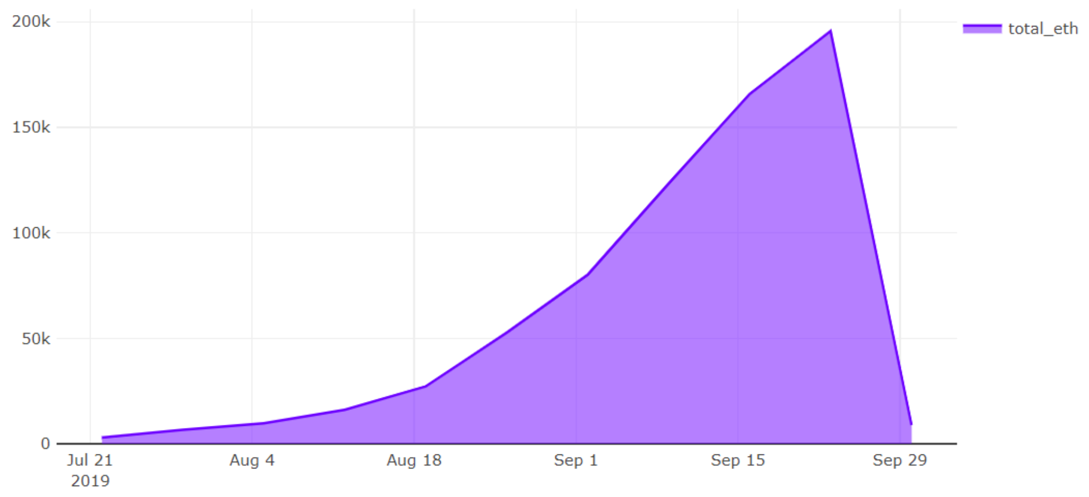
"Blockchain gives us confidence that smart contracts will operate as coded, but regular users can't always be confident they will operate as intended."
K. Petrie (7/2019)

Example: FairWin (9/2019)

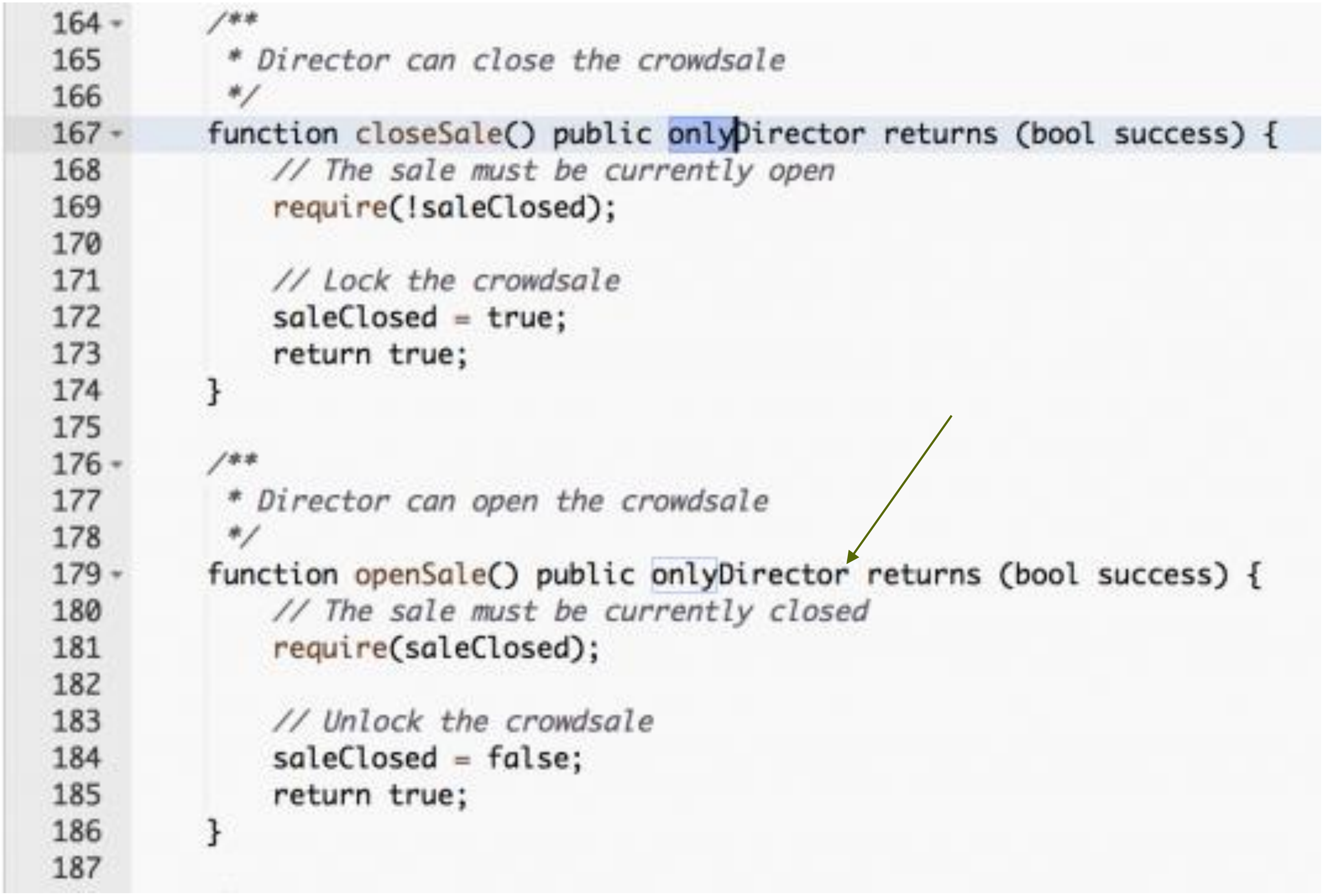
- "Vulnerabilities" that allow owner to totally drain balance, that allow owner to prevent users from withdrawing ETH forever, or that allows anyone to steal new deposits reported (9/27/2019)
  - *FairWin could "be one of the biggest scams ever seen in Ethereum."*

From $10M to Zero in 10 Days: ETH Smart Contract FairWin Is Empty

FairWin
0x01eacc3ae59ee7fbbc191d63e8e1ccfdac11628c



- Oyster Token (11/2018)
  - ICOs typically have one event to sell tokens
  - Oyster Token smart contract allows director to reopen



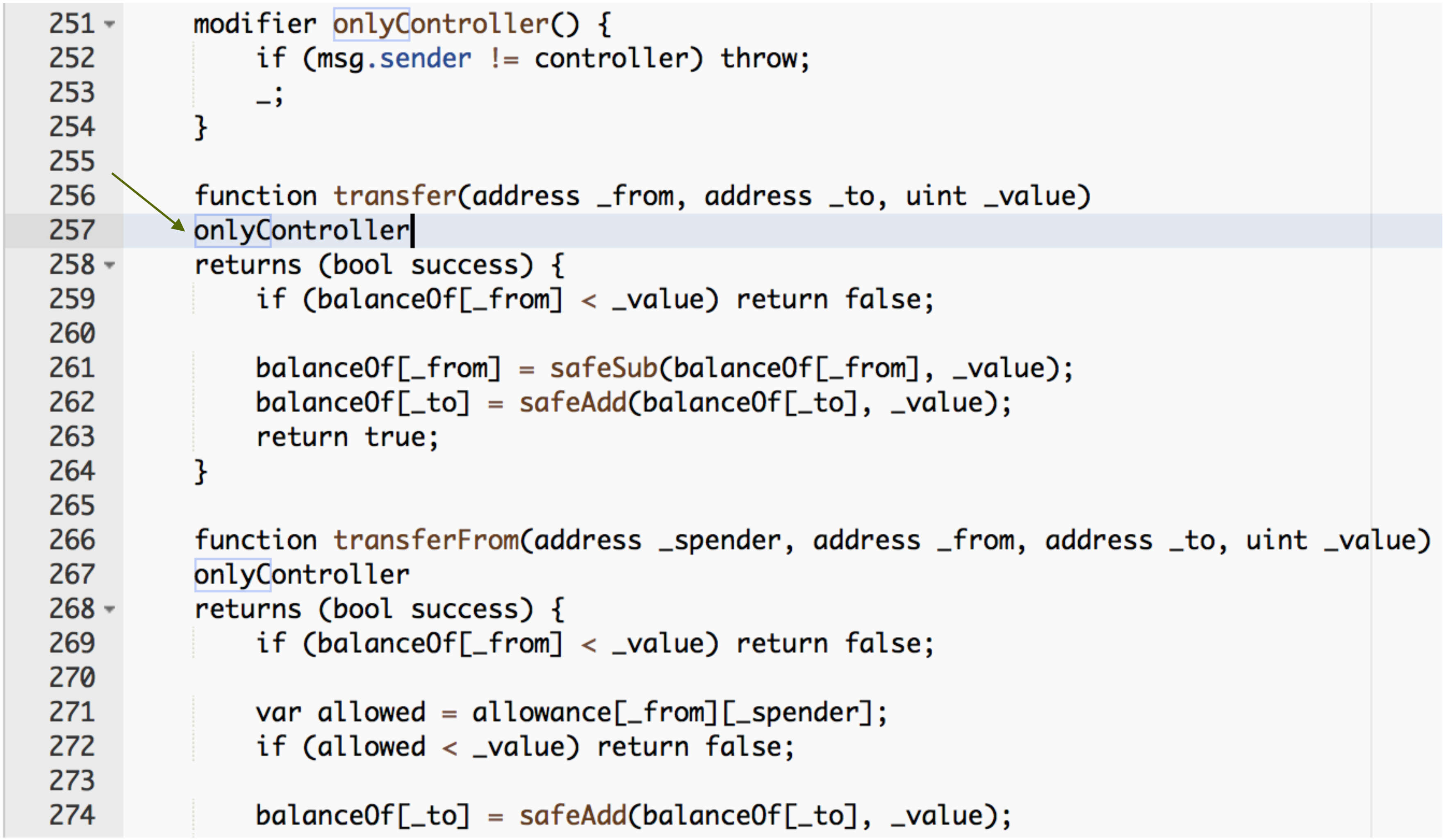
```
    /**
     * Director can close the crowdsale
     */
    function closeSale() public onlyDirector returns (bool success) {
        // The sale must be currently open
        require(!saleClosed);

        // Lock the crowdsale
        saleClosed = true;
        return true;
    }

    /**
     * Director can open the crowdsale
     */
    function openSale() public onlyDirector returns (bool success) {
        // The sale must be currently closed
        require(saleClosed);

        // Unlock the crowdsale
        saleClosed = false;
        return true;
    }
```

- FunFair token (11/2018)
  - Controller has ability to wipe out balance of contract (if hacked presumably, but even if not!)
  - Does a token purchaser have any recourse if it's in the contract code?



```
    modifier onlyController() {
        if (msg.sender != controller) throw;
        _;
    }

    function transfer(address _from, address _to, uint _value)
    onlyController
    returns (bool success) {
        if (balanceOf[_from] < _value) return false;

        balanceOf[_from] = safeSub(balanceOf[_from], _value);
        balanceOf[_to] = safeAdd(balanceOf[_to], _value);
        return true;
    }

    function transferFrom(address _spender, address _from, address _to, uint _value)
    onlyController
    returns (bool success) {
        if (balanceOf[_from] < _value) return false;

        var allowed = allowance[_from][_spender];
        if (allowed < _value) return false;

        balanceOf[_to] = safeAdd(balanceOf[_to], _value);
```

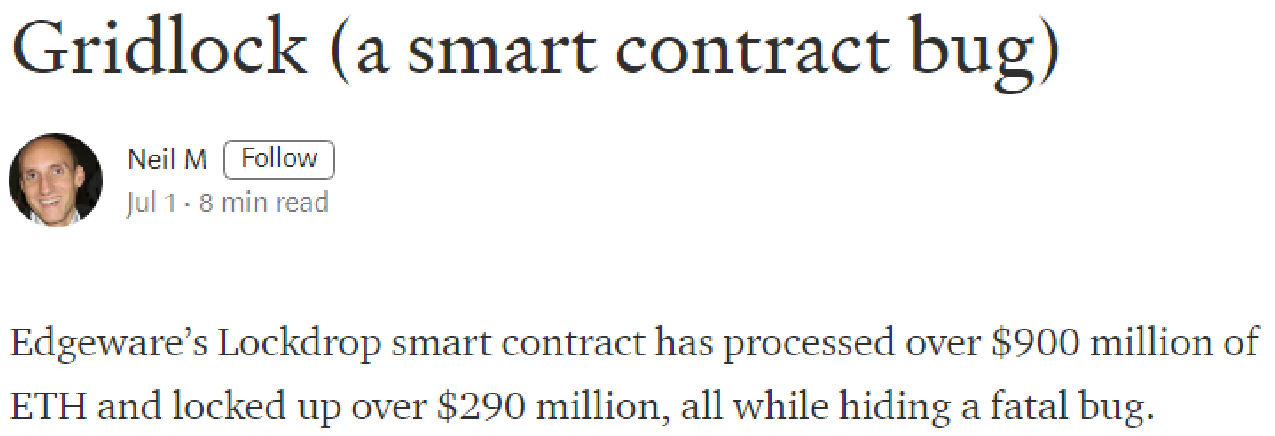
Incorrect assumptions

Initial contract state

- Contract authors assuming
  - No one knows contract addresses until they are created
  - Are initialized with no balance (e.g. hold no ETH)
  - Can only be sent ETH via payable functions including the fallback function

- But
  - Contract addresses predictable
    - Given the creator's address and nonce
    - Nonce starts at 1 and is incremented after each transaction from address
  - Contract addresses can be sent and have ETH associated with them *before* they are even created
  - Can be sent ETH via self-destruction of a contract

- Gridlock bug on Lockdrop contract (7/2019)



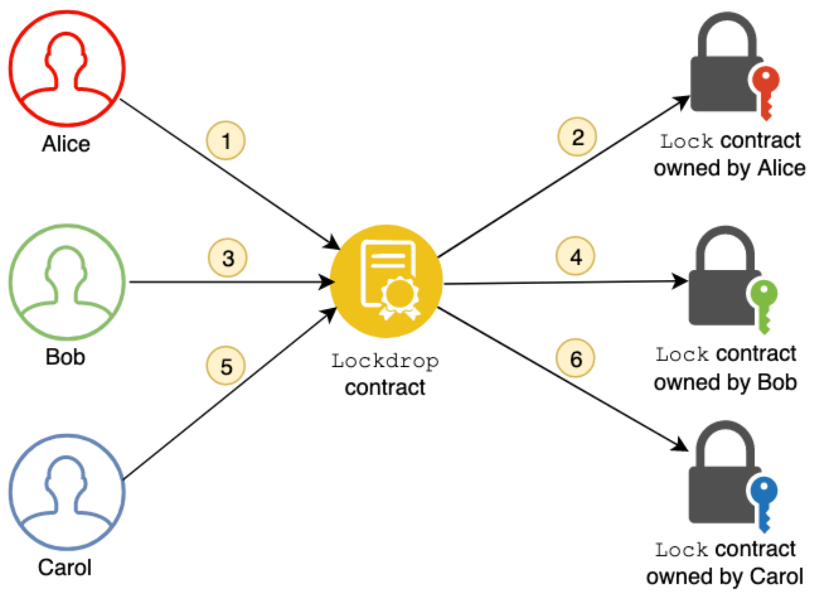
Gridlock (a smart contract bug)

Neil M Follow

Jul 1 · 8 min read

Edgeware's Lockdrop smart contract has processed over $900 million of ETH and locked up over $290 million, all while hiding a fatal bug.

- Fixed-size token purchase done in multiple steps
  - Wallets signal interest to buy
  - Wallets then commit ETH (steps 1, 3, 5) for 3-12 months in contract


Alice
Bob
Carol
1
3
5
Lockdrop contract
2
4
6
Lock contract owned by Alice
Lock contract owned by Bob
Lock contract owned by Carol

- Locks up ETH for the `msg.sender` (typically a smart contract) of the amount `msg.value`
  - Get `msg.value` amount of ETH from sender
  - Create a new contract using the ETH that locks it up for a period
  - After creation, ensure that the contract created has the correct balance

```
function lock(Term term, bytes calldata edgewareAddr, bool isValidator)
  external payable didStart didNotEnd {

  uint256 eth = msg.value;
  address owner = msg.sender;
  uint256 unlockTime = unlockTimeForTerm(term);
  // Create ETH lock contract
  Lock lockAddr = (new Lock).value(eth)(owner, unlockTime);
  // ensure lock contract has all ETH, or fail
  assert(address(lockAddr).balance == msg.value);
  emit Locked(owner, eth, lockAddr, term, edgewareAddr, isValidator, now);
}
```

  - `assert` assumes contract didn't receive any other ETH either before or after creation
    - Function is jammed forever if someone pre-sends ETH to address
    - Nonce only changes when a contract is successfully created
    - Assert will fail and roll-back results without advancing the nonce
    - Fix via

```
assert(address(lockAddr).balance >= msg.value);
```

Remediation

- Don't over-assert
- Remove any non-obvious behavior from the programming language and virtual machine
- Assume smart contracts will always contain bugs (unless proven otherwise)
- Audit via code analysis
  - Example: Slither's dangerous-strict-equality detector (Trail of Bits, crytic.io)