Blockchain and Cryptocurrencies

Week 11 - Smart Contracts Security

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2020

Sources/Background

- https://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/
- https://medium.com/@MyPaoG/explaining-the-dao-exploit-for-beginners-in-solidity-80ee84f0d470
- https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db35
- https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

Contents

Smart Contracts

- The realization of **smart contracts** begins with the Ethereum blockchain. 2015
- Lots of excitement DApps. etc.
- Lots of plans
- Emerging need for financing
- Enters the DAO

The Decentralized Autonomous Organization (DAO)

Est. April 2016

The Decentralized Autonomous Organization (DAO)

Est. April 2016

Purpose of the DAO

- advertising smart contract projects
- fund raising

[illegible]

The Decentralized Autonomous Organization (DAO)

Est. April 2016

Purpose of the DAO

[illegible]

- advertising smart contract projects
- fund raising

Working of the DAO

- proposers advertise projects
- proposal represented by contract
- investors vote for projects according to the size of their investment
- investment in terms of Ether
- implemented by smart contracts
  - first phase: collect funding and votes
  - second phase: pay out according to algorithm

Implementation of the DAO

- Christoph Jentzsch of Slock.It programmed the contracts needed
  https://github.com/slockit/DAO
- but he never put them on the blockchain!
- hoping for USD 500,000 in total

Execution

- 15% of **all** Ether in existence poured into the contract

Execution

- 15% of **all** Ether in existence poured into the contract
- net worth of USD 150,000,000

Execution

- 15% of **all** Ether in existence poured into the contract
- net worth of USD 150,000,000
- and then the DAO was hacked!

Execution

- 15% of **all** Ether in existence poured into the contract
- net worth of USD 150,000,000
- and then the DAO was hacked!
- money was extracted at a steady rate before the funding phase had started

Execution

- 15% of **all** Ether in existence poured into the contract
- net worth of USD 150,000,000
- and then the DAO was hacked!
- money was extracted at a steady rate before the funding phase had started
- ended by a hard fork (voted by a majority of the community)

Execution

- 15% of **all** Ether in existence poured into the contract
- net worth of USD 150,000,000
- and then the DAO was hacked!
- money was extracted at a steady rate before the funding phase had started
- ended by a hard fork (voted by a majority of the community)
  - ETH stopped all exchanges and miners, updated to new clients, and pretended the DAO never happened

Execution

- 15% of **all** Ether in existence poured into the contract
- net worth of USD 150,000,000
- and then the DAO was hacked!
- money was extracted at a steady rate before the funding phase had started
- ended by a hard fork (voted by a majority of the community)
    - ETH stopped all exchanges and miners, updated to new clients, and pretended the DAO never happened
    - ETC just carried on

A Little Background on the DAO Hack

- The DAO has a procedure to split to cater for minorities
- After some delay, a child DAO is created
- After more delay, one can transfer to a self-controlled account
- Attacker found a loophole
  - to retrieve the Ether first, and update the balance later
  - in between, recursively call the split procedure
  - effect: retrieve the funds multiple times

Technical Background on the DAO Hack

- An Ethereum contract may specify one nameless function, called **default function** or **fallback function**.
- It is invoked by all messages that do not match an existing function.
- For instance when transferring Ether to a **contract** (as opposed to a user account) without invoking a function explicitly, then the fallback is called.

```
contract Sink {
    function() external payable { }
}
```

VulnerableContract

```
function getBalance(address user) constant returns(uint) {
  return userBalances[user];
}

function addToBalance() {
  userBalances[msg.sender] += msg.amount;
}

function withdrawBalance() {
  amountToWithdraw = userBalances[msg.sender];
  if (!(msg.sender.call.value(amountToWithdraw)())) { throw; }
  userBalances[msg.sender] = 0;
}
```

- msg.sender address of sender
- msg.amount Ether that comes with the message
- .value(...) adds Ether to a method call
- what can go wrong?

The Attack: race-to-empty wallet

- The code invokes the fallback function of msg.sender
- An attacker might set up the following to withdraw twice the balance

```
VulnerableContract v;
uint times = 0;
...
v.withdrawBalance();
...
function () {
  // To be called by a vulnerable contract with a withdraw function.
  // This will double withdraw.

  if (times == 0) {
    times = 1;
    v.withdrawBalance();

   } else { times = 0; }
}
```

Mitigation 1

Use correct ordering (requires careful code rewrite)

```
function withdrawBalance() {
  amountToWithdraw = userBalances[msg.sender];
  userBalances[msg.sender] = 0;
  if (amountToWithdraw > 0) {
    if (!(msg.sender.send(amountToWithdraw))) { throw; }
  }
}
```

- On the second round, the attacker will see balance zero and obtain no further funds.

Mitigation 2

Insert "mutex" (requires stylized code rewrite)

```
function withdrawBalance() {
  if ( withdrawMutex[msg.sender] == true) { throw; }
  withdrawMutex[msg.sender] = true;
  amountToWithdraw = userBalances[msg.sender];
  if (amountToWithdraw > 0) {
    if (!(msg.sender.send(amountToWithdraw))) { throw; }
  }
  userBalances[msg.sender] = 0;
  withdrawMutex[msg.sender] = false;
}
```

- second call will fail
- disadvantage: higher gas use

Contents

King of the Ether

What is KotE?

An Ethereum contract, living on the blockchain, that will make you a King or Queen, might grant you riches, and will immortalize your name.

https://www.kingoftheether.com/

King of the Ether

What is KotE?

An Ethereum contract, living on the blockchain, that will make you a King or Queen, might grant you riches, and will immortalize your name.
https://www.kingoftheether.com/

Word of Caution!

A classical Ponzi scheme with a twist

KotE Contract Rules

- To become Monarch, you send the current price for the throne, 10 ETHER, say, to the contract.

KotE Contract Rules

- To become Monarch, you send the current price for the throne, 10 ETHER, say, to the contract.
- You become the illustrious new King of the Ether. Your name will be added to the Hall of Monarchs in the blockchain.

KotE Contract Rules

- To become Monarch, you send the current price for the throne, 10 ETHER, say, to the contract.
- You become the illustrious new King of the Ether. Your name will be added to the Hall of Monarchs in the blockchain.
- The contract will send your 10 ETHER (less a small commission) to the previous Monarch.

KotE Contract Rules

- To become Monarch, you send the current price for the throne, 10 ETHER, say, to the contract.
- You become the illustrious new King of the Ether. Your name will be added to the Hall of Monarchs in the blockchain.
- The contract will send your 10 ETHER (less a small commission) to the previous Monarch.
- The price for the throne will go up by 33%, to 13.3 ETHER.

KotE Contract Rules

- To become Monarch, you send the current price for the throne, 10 ETHER, say, to the contract.
- You become the illustrious new King of the Ether. Your name will be added to the Hall of Monarchs in the blockchain.
- The contract will send your 10 ETHER (less a small commission) to the previous Monarch.
- The price for the throne will go up by 33%, to 13.3 ETHER.
- The difference will be your profit if the next usurper buys the throne.

KotE Contract Rules

- To become Monarch, you send the current price for the throne, 10 ETHER, say, to the contract.
- You become the illustrious new King of the Ether. Your name will be added to the Hall of Monarchs in the blockchain.
- The contract will send your 10 ETHER (less a small commission) to the previous Monarch.
- The price for the throne will go up by 33%, to 13.3 ETHER.
- The difference will be your profit if the next usurper buys the throne.
- However, an ancient curse applies to the throne: every Monarch dies once their reign reaches 14 days. No compensation is paid if this happens, and the claim price for the next monarch is reset back to the original starting price of 500 FINNEY (0.5 ETHER).

KotE Contract Rules

- To become Monarch, you send the current price for the throne, 10 ETHER, say, to the contract.
- You become the illustrious new King of the Ether. Your name will be added to the Hall of Monarchs in the blockchain.
- The contract will send your 10 ETHER (less a small commission) to the previous Monarch.
- The price for the throne will go up by 33%, to 13.3 ETHER.
- The difference will be your profit if the next usurper buys the throne.
- However, an ancient curse applies to the throne: every Monarch dies once their reign reaches 14 days. No compensation is paid if this happens, and the claim price for the next monarch is reset back to the original starting price of 500 FINNEY (0.5 ETHER).
- But surely a successor will appear within 14 days, and even if they don't, you'll be immortalised in the blockchain at least . . .

Issue

A version of KotE had a bug (similar to Listing 2 below) that prevented correct payment.

What can go wrong?

Consider the end of a game

```
/*** Listing 1 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  winner.send(1000); // send a prize to the winner
  prizePaidOut = True;
}
```

Explanation

- winner is an address
- send method transfers tokens

What can go wrong?

Consider the end of a game

```
/*** Listing 1 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  winner.send(1000); // send a prize to the winner
  prizePaidOut = True;
}
```

Explanation

- winner is an address
- send method transfers tokens

What if send fails?

It just returns false. . .

1. winner may be a contract and sending may fail for lack of gas or due to an exception
2. sending may fail because of callstack overflow

In both cases, the code continues and sets prizePaidOut to True.

Amendment

From Ethereum documentation

There are some dangers in using send: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of send or even better: Use a pattern where the recipient withdraws the money.

Amendment

From Ethereum documentation

There are some dangers in using send: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of send or even better: Use a pattern where the recipient withdraws the money.

Revised code

```
/*** Listing 2 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  if (winner.send(1000))
    prizePaidOut = True;
  else throw;
}
```

Works ok

Modifying the Requirements

Suppose we also want to give a litte to the loser

```
/*** Listing 3 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  if (winner.send(1000) && loser.send(10))
    prizePaidOut = True;
  else throw;
}
```

- protects from callstack overflow
- but introduces dependency between winner and loser

Better Approach

```
/*** Listing 4 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
  if (callStackIsEmpty()) throw;
  winner.send(1000)
  loser.send(10)
  prizePaidOut = True;
}
```

- callStackIsEmpty performs a test call to check whether the call stack is exhausted
- If it is, the exception stops with an exception and reverts all changes effected by the contract
- any other failure is contained within winner or loser

Pattern Where Recipient Withdraws the Money

Instead of sending tokens, the contract sets up an entrypoint where recipients can pick up their tokens.

```
/*** Listing 5 ***/
if (gameHasEnded && !( prizePaidOut ) ) {
   accounts[winner] += 1000
   accounts[loser] += 10
   prizePaidOut = True;
}
...
function withdraw(amount) {
   if (accounts[msg.sender] >= amount) {
      accounts[msg.sender] -= amount;
      msg.sender.send(amount);
   }
}
```

KotE Resolution

The "King of the Ether Throne" lottery game is the most well-known case of this bug so far. This bug wasn't noticed until after a sum of 200 Ether (worth more than USD 2000 at today's price) failed to reach a rightful lottery winner. The relevant code in King of the Ether resembles that of Listing 2. Fortunately, in this case, the contract developer was able to use an unrelated function in the contract as a "manual override" to release the stuck funds. A less scrupulous administrator could have used the same function to steal the Ether!
Almost a year earlier (while Ethereum was in its "frontier" release), a popular lottery contract, EtherPot [9], also suffered from the same bug.
An earlier version of BTCRelay also exhibited this bug [7]. Although the hazard was noticed in an earlier security audit, the wrong fix was applied at first [8].

https://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/

Contents

An Incident

In July 2017, . . .

the amount of 153,037 Ether (worth more than USD 30M) was stolen from three large Ethereum multisig wallet contracts.
The MultisigExploit-Hacker (MEH) exploited a vulnerability in the Parity 1.5 client's multisig wallet contract.
MEH took ownership of a victim's wallet with a single transaction.
The attacker could then drain the victim's funds.
The victims were three ICO projects: Edgeless Casino, Swarm City, and æternity

Source `https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`

How the Attack Worked

- instructive
- further insight into the working of Ethereum

Contract Libraries

- Ethereum contracts can be split into multiple parts

⇒ libraries

- avoids code duplication (which saves gas)
- libraries can be updated without changing the address of the importing contract

⇒ flexibility, upgrades and bug fixes possible

The Wallet Library

```
contract WalletLibrary {
    address owner;

    // called by constructor
    function initWallet(address _owner) {
        owner = _owner;
        // ... more setup ...
    }

    function changeOwner(address _new_owner) external {
        if (msg.sender == owner) {
            owner = _new_owner;
        }
    }

    function () payable {
        // ... receive money, log events, ...
    }

    function withdraw(uint amount) external returns (bool success) {
        if (msg.sender == owner) {
            return owner.send(amount);
        } else {
            return false;
} } }
```

Standard wallet functionality

- initializer sets the owner
- the owner can set a new owner
- everyone can deposit
- only owner can withdraw

The Wallet Contract

```
contract Wallet {
    address _walletLibrary;
    address owner;

    function Wallet(address _owner) {
        _walletLibrary = <address of pre-deployed WalletLibrary>;
        _walletLibrary.delegatecall(bytes4(sha3("initWallet(address)")), _owner);
    }

    function withdraw(uint amount) returns (bool success) {
        return _walletLibrary.delegatecall(bytes4(sha3("withdraw(uint)")), amount);
    }

    // fallback function gets called if no other function matches call
    function () payable {
        _walletLibrary.delegatecall(msg.data);
    }
}
```

Analysis

- The Wallet holds two variables
  - _walletLibrary is assumed to hold the address of a deployment of the WalletLibrary given a suitable method, this address can be changed!
  - owner address of the owner, to be set in the constructor
- function Wallet is the constructor. Called when a new instance of Wallet is created.
- Most functionality is implemented by delegating to WalletLibrary

How Delegation works

- Each contract maintains a table of external entry points
- Indexed by SHA3 hash of name and signature of the entry point (because Solidity admits overloading)
- the delegatecall method takes such a hash and further arguments and calls the method via this index
- hence, the constructor calls the initWallet method of the library
- Important: library code runs in the storage frame of the caller!
- the fallback function delegates to the library by passing msg.data
- this forwarder enables calling **any** function in the library, unless the call is already handled by Wallet

How the Attack Works

- The attacker calls initWallet with its own address on a Wallet contract

How the Attack Works

- The attacker calls initWallet with its own address on a Wallet contract
- As Wallet has no matching method, the fallback is invoked.

How the Attack Works

- The attacker calls initWallet with its own address on a Wallet contract
- As Wallet has no matching method, the fallback is invoked.
- The fallback delegates to WalletLibrary

How the Attack Works

- The attacker calls initWallet with its own address on a Wallet contract
- As Wallet has no matching method, the fallback is invoked.
- The fallback delegates to WalletLibrary
- . . . which changes the owner of the Wallet

How the Attack Works

- The attacker calls initWallet with its own address on a Wallet contract
- As Wallet has no matching method, the fallback is invoked.
- The fallback delegates to WalletLibrary
- . . . which changes the owner of the Wallet
- Next, the attacker can withdraw everything!

How the Attack Could Have Been Prevented

- Solidity can mark functions as internal or external
- An internal initWallet function would not show up in the table of entry points
- A same-named function in Wallet would shadow the library function.

Thanks!