Lecture 03:
Smart Contract Security

Akash Khosla

LECTURE OUTLINE

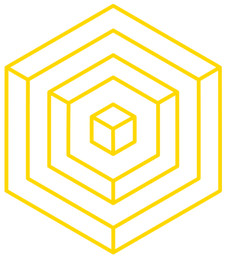
1. SECURITY OVERVIEW
2. SMART CONTRACT SECURITY
3. SOME QUIRKS
4. HACKING HISTORY (EXTRA)

1 SECURITY OVERVIEW

WHAT SECURITY MEANS FOR BLOCKCHAIN

GENERAL OVERVIEW

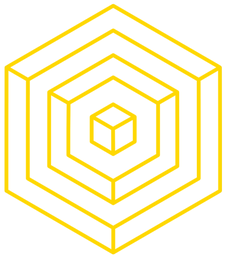
- Security refers to preventing exploit on the protocol or application layers
- What's so bad about a **public blockchain**?
  - Think of the blockchain as a server, where server side code is completely exposed to the **clients on the protocol (network nodes)** and **clients on DApps (users)**
  - It's all public - therefore imperative that secure code is written
  - Imagine taking your private, permissioned database and having to make it public, and distributed, yet secure
- Big tradeoff of Ethereum vs. Bitcoin -> **Turing Completeness of the EVM**
  - Attack surface is a problem on Ethereum



WHAT IS THE REALITY

GENERAL OVERVIEW

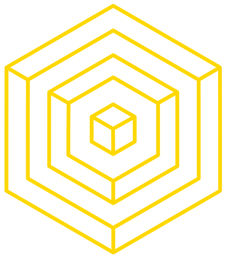
- Ethereum and other complex blockchain systems are highly experimental
  - Requires a new philosophy behind development
  - Unstable with a lot of potential to go wrong
    - Like programming hardware or financial services rather than mobile/web
- **People can drain millions from contract accounts**
- Several exploits from on even the best of the best teams in the space
- Suppose people have their **own instances of a contract deployed**, i.e. multi-sig wallet
  - There's no built in way to govern that each person upgrade or `selfdestruct` into a new instance once an exploit happens
  - Must write robust code **prior to mainnet deployment**



INTRODUCING SMART CONTRACT SECURITY

GENERAL OVERVIEW

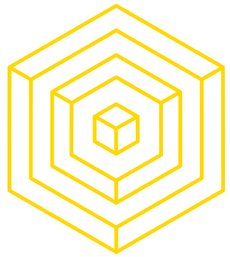
- Smart contracts are **immutable**
  - Once deployed, you cannot change their code
  - Therefore you **cannot fix the discovered bugs** in the contract
  - Compare to current state of software engineering
    - Continuous Deployment, Integration Tests, etc.
- The future might be whole organizations governed by smart contract code
  - So there's clearly a need for proper security
- *"Disneyland for Hackers"*



WHAT'S THE SOLUTION

GENERAL OVERVIEW

- Better testing methods for smart contracts
  - TDD is often argued with as it's never a guarantee of security, but then again, nothing is **truly** secure
- Formally verifiable smart contracts
  - Mathematically prove why the smart contract works
- Smart contract code audits
  - Get multiple sources to audit code before going live
- Easier smart contract languages with safety built in
- Have very careful programmers write code on this platform



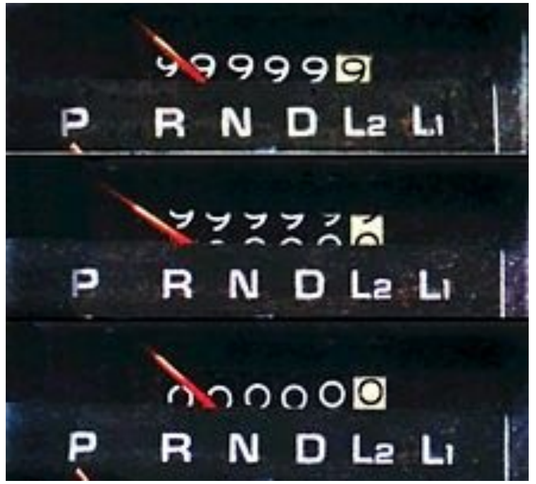
2 SMART CONTRACT SECURITY

INTEGER OVERFLOW

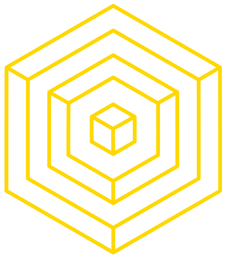
GENERAL OVERVIEW

- Suppose you increment a number above it's **max value**
  - Solidity can handle up to 256 bit numbers (2**256 - 1)
  - **Overflow**: Incrementing the number by 1 would result in 0

```
  0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
+ 0x0000000000000000000000000000000000000000000000000000000001
------------------------------------------------------------
= 0x0000000000000000000000000000000000000000000000000000000000
```



After reaching the maximum reading, an odometer or trip meter restarts from zero, called odometer rollover.



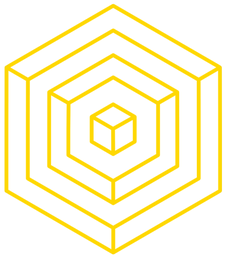
INTEGER UNDERFLOW

GENERAL OVERVIEW

- Likewise, in the inverse case, when the number is unsigned, decrementing will **underflow** the number, resulting in the **maximum possible value**

```
  0x0000000000000000000000000000000000000000
- 0x0000000000000000000000000000000000000001
------------------------------------------
= 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Demo

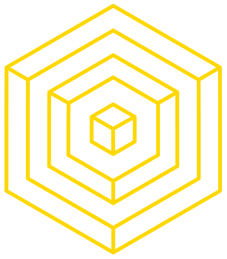


INTEGER OVERFLOW AND UNDERFLOW

GENERAL OVERVIEW

- Both cases are very dangerous, but the underflow case is the more likely one to happen
- Suppose a token holder has **X** tokens but attempts to spend **X+1**
- If the code doesn't check for it, the attack may end up being allowed to spend more tokens than he had and his balance **underflows** to the **max integer**
- **Mitigation**: OpenZeppelin's Safemath library

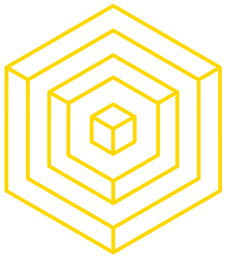
Demo



VISIBILITY

GENERAL OVERVIEW

- **`Public`** functions can be called by anyone
  - By functions inside the contract, functions from inherited contracts or outside users
- **`External`** functions can only be accessed externally by other contracts (Demo)
  - They cannot be called by other functions of the contract/not visible to the contract itself
  - Cheaper to use because it uses the **`calldata`** opcode while public needs to copy all arguments to memory (source)
- **`Private`** functions can be called only from inside the contract
- **`Internal`** functions are a more relaxed version of **`private`**, where contracts that inherit from the parent are able to use the the function

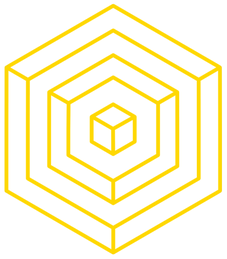


VISIBILITY

GENERAL OVERVIEW

- **`Public`** functions can be called by anyone
  - By functions inside the contract, functions from inherited contracts or outside users
- **`External`** functions can only be accessed externally by other contracts (Demo)
  - They cannot be called by other functions of the contract/not visible to the contract itself
  - Cheaper to use because it uses the **`calldata`** opcode while public needs to copy all arguments to memory (source)
- **`Private`** functions can be called only from inside the contract
- **`Internal`** functions are a more relaxed version of **`private`**, where contracts that inherit from the parent are able to use the the function

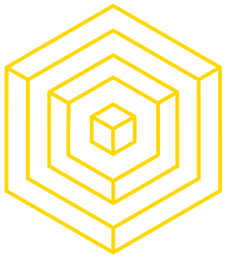
Conclusion: Keep your functions **`private`** or **`internal`** unless there is a need for outside interaction.



FALLBACK FUNCTION

GENERAL OVERVIEW

- A contract can have exactly one unnamed function
  - This function cannot have arguments and cannot return anything
- The function is executed on a call to the contract **if none of the other functions match the given function identifier** (or if no data was supplied at all)
- To receive ether for this function, we must mark it as **`payable`**
- Even though the fallback function cannot have arguments, one can still use **`msg.data`** to retrieve any payload supplied with the call



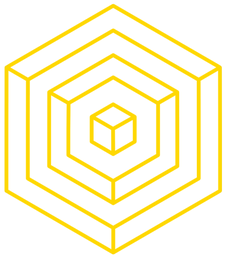
DELEGATECALL

GENERAL OVERVIEW

- **`delegatecall`** is identical to a message call (internal transaction) apart from the fact that the code at target address is executed in the context of the calling contract and **`msg.sender`** and **`msg.value`** do not change their values
  - A contract can dynamically load code from a different address at runtime with **`delegatecall`**
  - Storage, current address and balance still refer to the calling contract, only the code is taken from the address we called
- Very useful for implementing libraries and modularizing code
- But opens doors to vulnerabilities as your contract is allowing anyone to do whatever they want with their state

```
contract D {
    uint public n;
    address public sender;

    function delegatecallSetN(address _e, uint _n) {
      _e.delegatecall(bytes4(sha3(setN(uint256))), _n); /* D's storage is set, E is not modified
*/
    }
  }

  contract E {
    uint public n;
    address public sender;
    function setN(uint _n) {
      n = _n;
      sender = msg.sender;
    }
  }
```

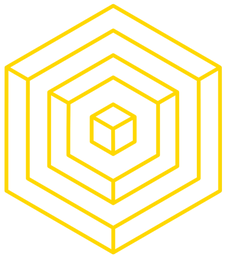
AUTHOR: AKASH KHOSLA



ASIDE ON PARITY HACK

DANGERS OF DELEGATECALL

- The Parity hack involved a combination of both insecure visibility modifiers and misuse of **`delegatecall`** with arbitrary data
- The vulnerable contract's function implemented **`delegatecall`** and a function from another contract that could modify ownership was left public
- That allowed an attacker to craft the **`msg.data`** field to call the vulnerable function
  - As for what would be included in the **`msg.data`** field, that is the signature of the function that you want to call
  - Signature here means the first 8 bytes of the sha3 (alias for keccak256) hash of the function prototype

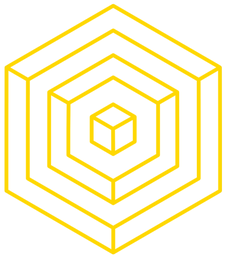


ASIDE ON PARITY HACK

DANGERS OF DELEGATECALL

- That allowed an attacker to craft the **`msg.data`** field to call the vulnerable function
  - As for what would be included in the **`msg.data`** field, that is the signature of the function that you want to call
  - Signature here means the first 8 bytes of the **`sha3`** (alias for keccak256) hash of the function prototype

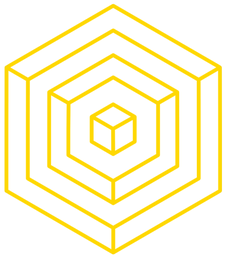
```
// In this case:
web3.sha3("pwn()").slice(0, 10) --> 0xdd365b8b
// If the function takes an argument, pwn(uint256 x):
web3.sha3("pwn(uint256").slice(0,10) --> 0x35f4581b
```



CALL COMPARISON

GENERAL OVERVIEW

- **`delegatecall`** says “I’m a contract and I’m allowing (delegating) you to do whatever you want to my storage”
  - Security risk for the sending contract which needs to trust that the receiving contract will treat the storage well
- **`callcode`** does the same thing, but it did not preserve **`msg.sender`** and **`msg.value`**
  - If Alice invokes Bob who does **`delegatecall`** to Charlie, the msg.sender in the **`delegatecall`** is Alice
  - If **`callcode`** was used the **`msg.sender`** would be Bob
- **`call`** is the basic way to call code from other contracts, except state used is within the called contract
  - When contract D does a **`call`** on contract E, the code runs in the context of E: the storage of E is used.



REENTRANCY ATTACK

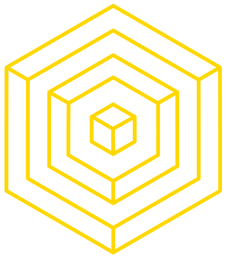
THE DAO HACK

- Solidity's `call` function when called with `value` forwards all the gas it received
- In the snippet below, the call is made before actually reducing the sender's balance:

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

AUTHOR: AKASH KHOSLA

SOURCE



REENTRANCY ATTACK

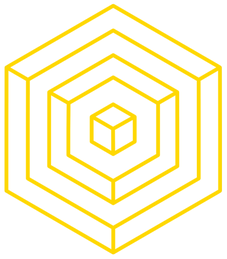
THE DAO HACK

“In simple words, it’s like the bank teller doesn’t change your balance until she has given you all the money you requested. Can I withdraw $500? Wait, before that, can I withdraw $500? And so on. The smart contracts as designed only check you have $500 at the beginning, once, and allow themselves to be interrupted.” - Wise Reddit Commenter

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

AUTHOR: AKASH KHOSLA

SOURCE

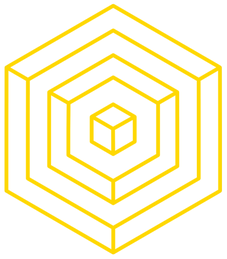


REENTRANCY ATTACK

THE DAO HACK

- Solution 1: Reduce the sender's balance **before** making the transfer of value
- Solution 2: Use mutexes to mitigate race conditions
- Solution 3 (Best): use **`require(msg.sender.transfer(_value))`**
  - Read more on about benefits and tradeoffs on Stack Exchange
  - Revert vs. Require vs. Assert

AUTHOR: AKASH KHOSLA
SOURCE



FORCE ETHER TO A CONTRACT

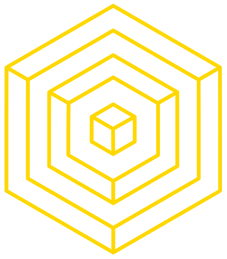
SELFDESTRUCT

Solidity's **`selfdestruct`** does two things:

1. It renders the contract useless, effectively deleting the bytecode at that address
2. It sends all the contract's funds to a target address

The special case here, is that if the receiving address is a contract, its fallback function does not get executed

If a contract's function has a conditional statement that depends on that contract's balance being below a certain amount, that statement can be potentially bypassed



FORCE ETHER TO A CONTRACT

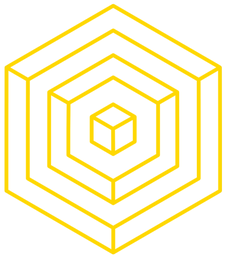
EXAMPLE

```
pragma solidity 0.4.18;

contract ForceEther {

 bool youWin = false;

 function onlyNonZeroBalance() {
     require(this.balance > 0);
     youWin = true;
 }
 // throw if any ether is received
 function() payable {
   revert();
 }

}
```

Due to the throwing fallback function, normally the contract cannot receive ether. However, if a contract **`selfdestructs`** with this contract as a target, the fallback function does not get called.

As a result **`this.balance`** becomes greater than 0, and thus the attacker can bypass the **`require`** statement in **`onlyNonZeroBalance`**

Fix: Never use a contract's balance as a guard.



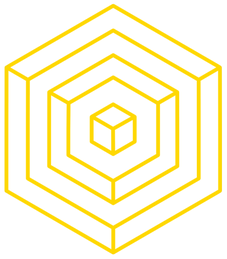
DoS WITH UNEXPECTED REVERT

KING OF THE ETHER

- You want to be the king of the throne in this contract? Here are the rules:
  - If the current throne price is 10 ETH, you send 10 ETH to be the king
  - You will be added to the Hall of Monarchs on the blockchain
  - The contract will then send your 10 ETH (a little less due to commission) to the previous monarch you usurped as compensation
  - Price to claim the throne goes up by 33% to 13.3 ETH (stops at 100000 ETH for safety)
    - If someone comes along who is will to pay 13.3 ETH, you'll make a profit from this!
  - Every monarch dies once their reign reaches 14 days, no compensation is paid if this happens, and the claim price is reset back to the original starting price of 0.5 ETH.

AUTHOR: AKASH KHOSLA

SOURCE

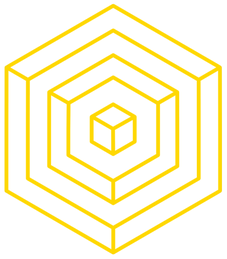


DoS WITH UNEXPECTED REVERT

CALL TO THE UNKNOWN

- DEMO
- In this case, an attacker's contract could first claim leadership by sending enough ether to the insecure contract. Then, the transactions of another player who would attempt to claim leadership would throw.
- Although a simple attack, this causes a permanent denial of service to the contract rendering it useless. This can be found in other ponzi scheme contracts that follow the same pattern.

AUTHOR: AKASH KHOSLA
SOURCE



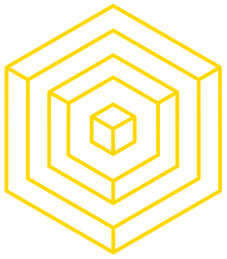
SHORT ADDRESS ATTACK

MAKE 10M FROM BLOCKCHAIN HACKS

- Allows an attacker to abuse the transfer function of an ERC20 (standard token contract) to withdraw a larger amount than he is allowed to
- Suppose we have an exchange with a wallet of **1000 tokens** (ERC-20 spec) and a user with a balance of **32 tokens** on that exchange's wallet
- **User address:** 0x12345600 - notice the trailing zeroes
- Suppose the user wants to withdraw an amount larger than their balance
  - They go to the exchange, click the token's withdraw button and input their address without the trailing zeroes (exchange does not perform input validation and let's the txn go through despite invalid address length)

AUTHOR: AKASH KHOSLA

SOURCE



SHORT ADDRESS ATTACK

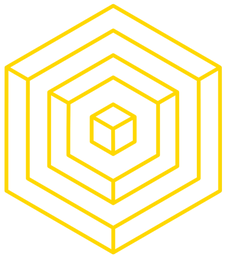
MAKE 10M FROM BLOCKCHAIN HACKS

The EVM calculates the input data for the transaction by concatenating the function signature and the arguments. The ERC20's transfer function is formulated as transfer(address to, uint256 amount). The 3 fields would be as follows:

```
sig : a9059cbb = web3.sha3("transfer(address,uint256)").slice(0,10)
arg1: 123456   = receiving address
arg2: 00000020  = 32 in hexademical (0x20)
---------------------------------------
Concatenated: a9059cbb 123456 00000020
Transaction input data: 0xa9059cbb12345600000020
```

AUTHOR: AKASH KHOSLA

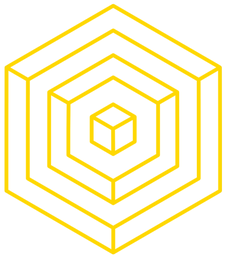
SOURCE



SHORT ADDRESS ATTACK

MAKE 10M FROM BLOCKCHAIN HACKS

```
sig : a9059cbb = web3.sha3("transfer(address,uint256)").slice(0,10)
arg1: 123456   = receiving address
arg2: 00000020  = 32 in hexademical (0x20)
---------------------------------------
Concatenated: a9059cbb 123456 00000020
Transaction input data: 0xa9059cbb12345600000020
// EVM pads the transaction with trailing zeros since 2 bytes shorter
0xa9059cbb1234560000002000
// a9059cbb = web3.sha3("transfer(address,uint256)").slice(0,10)
// 12345600   = receiving address
// 00002000  = 8192 in hexademical (0x2000 == 0x20<<8) // Damn, he's rich
```



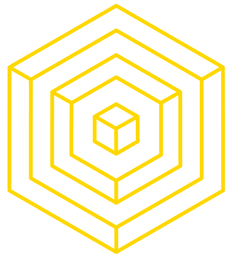
SHORT ADDRESS ATTACK

MAKE 10M FROM BLOCKCHAIN HACKS

- How to mitigate:
  - Throw if msg.data has invalid size
  - Exchanges must perform input validation

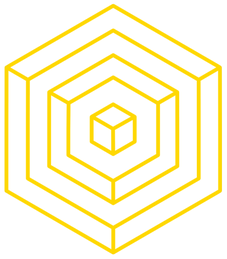
AUTHOR: AKASH KHOSLA

SOURCE



3

SOME SOLIDITY QUIRKS

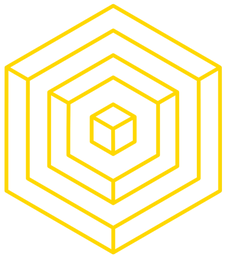


RANDOMNESS

DIFFICULT TO ACHIEVE

- Everything on the EVM is deterministic
- People have tried to use now and block.blockhash for business logic in a contract
  - However, results are predictable and can be manipulated by miners
  - **“Timestamp dependence”**
- Block numbers and average block time can be used to estimate time, but this is not future proof as block times may change (such as the changes expected during Casper).
- How can I securely generate a random number in my smart contract?

AUTHOR: AKASH KHOSLA
SOURCE



WHAT CAN I DO TOWARDS SECURITY

FOR SMART CONTRACTS

- Don’t write fancy code
- Use audited and tested code
- Write as many unit tests as possible
- Prepare for failure - at any moment, in any contract or method
- Rollout carefully - bug bounties before ICO
- Keep contracts simple - more complexity = more attack vectors
- Keep updating with software and community
- Beware of blockchain properties: public vs. private, .send() vs. .call()

AUTHOR: AKASH KHOSLA
SOURCE

EXTERNAL CALLS

Takeaway: Avoid external calls when possible

- What is an external call?
  - A call from one contract to another untrusted contract or account.
  - **delegatecall, callcode, call**
- Attack Possibility:
  - execute malicious code in that contract or any other contract that it depends upon.
- Previous (Expensive) Bugs:
  - The Dao hack
  - The Parity multisignature wallet hack

EXTERNAL CALLS: .send() vs .call.value()

Takeaway: use .send() and .transfer() over .call.value()

- .send()
  - Usage: `someaddress.send( ether )`
  - Description: sends ether from contract to *someaddress*, returns boolean of success
  - Security: triggers code execution, the called contract is only given a stipend of 2,300 gas which is currently only enough to log an event.
- .transfer()
  - Usage: `someaddress.transfer( ether )`
  - Description: equivalent to `if ( !someaddress.send( ether ) ) throw;`
  - Security: equivalent security to .send() with added failure protection

AUTHOR: COLLIN CHIN
https://github.com/ConsenSys/smart-contract-best-practices

EXTERNAL CALLS: .send() vs .call.value()

Takeaway: use .send() and .transfer() over .call.value()

- .call.value()
  - Usage: *someAddress*.call.value(*ether*)()
  - Description: calls *someAddress* address and sends ether.
  - Security: The executed code is given all available gas for execution making this type of value transfer unsafe against reentrancy.

REVERT VS THROW

Different bytecode. From the docs:

> `The throw keyword can also be used as an alternative to revert().`

Starting with Metropolis, `revert();` will return unused gas whereas `throw` will continue to consume all available gas. In terms of contract state changes, they are the same - everything gets rolled back.

(Source)

ASSERT AND REQUIRE

Takeaway: Use assert and require properly

- `require(condition)` is meant to be used for input validation
  - Should be done on any user input, and throws if condition is false.
- `assert(condition)` also throws if condition is false
  - Should be used only for internal errors or to check if your contract has reached an invalid state.
- By following this paradigm it would become possible to use a formal analysis tool, that verifies that the invalid opcode can never be reached, for the absence of errors assuming valid inputs.

AUTHOR: COLLIN CHIN
https://github.com/ConsenSys/smart-contract-best-practices

PRAGMA STATEMENTS

Takeaway: Lock pragmas to specific compiler version

```
// bad
pragma solidity ^0.4.4;


// good
pragma solidity 0.4.4;
```

AUTHOR: COLLIN CHIN
https://github.com/ConsenSys/smart-contract-best-practices

BLOCKCHAIN
AT BERKELEY

ROUNDING WITH INT DIVISION

Takeaway: All integer division rounds down to the nearest integer.
If you need more precision, consider using a multiplier, or store both the numerator and denominator.

```
// bad
uint x = 5 / 2; // Result is 2, all integer division rounds DOWN to the nearest integer


// good
uint multiplier = 10;
uint x = (5 * multiplier) / 2;


uint numerator = 5;
uint denominator = 2;
```

ALL DATA IS PUBLIC

Takeaway: Remember that on-chain data is public

- Many applications require submitted data to be private up until some point in time in order to work.
- If you are building an application where privacy is an issue, take care to avoid requiring users to publish information too early.

Example: Rock - Paper - Scissors

1. require both players to submit a hash of their intended move first
2. require both players to submit their move;
3. if the submitted move does not match the hash throw it out.

AUTHOR: COLLIN CHIN
https://github.com/ConsenSys/smart-contract-best-practices

2-PARTY / N-PARTY CONTRACTS

Takeaway: beware of the possibility that some participants may "drop offline" and not return

- Do not make refund or claim processes dependent on a specific party performing a particular action with no other way of getting the funds out.

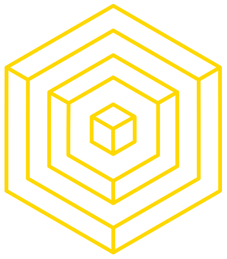
Example: Rock - Paper - Scissors

- Common Mistake: Not make a payout until both players submit their moves;
- Pitfall: a malicious player can "grief" the other by simply never submitting their move
- Solutions:
    1. Provide a way of circumventing non-participating participants. Ex: time limit
    2. Add an additional economic incentive for participants to submit information in all of the situations in which they are supposed to do so. Ex: deposit stake and slash

TEST COVERAGE

- Test driven development
- Go for 💯 💯 💯 💯 💯 💯 💯 💯 💯 test coverage
  - Test every method
  - Prevent unintended behavior

```
Compiling ConvertLib.sol...
Compiling MetaCoin.sol...
Compiling Migrations.sol...


  Contract: MetaCoin
    ✓ should put 10000 MetaCoin in the first account (107ms)
    ✓ should call a function that depends on a linked library (109ms)
    ✓ should send coin correctly (153ms)


  3 passing (4s)
```



TOOLS FOR AUDITING AND ANALYSIS

FOR SMART CONTRACTS

First of all, solc performing semantic checking is a huge step towards security as potential mistakes get found at compilation time.

- Securify.ch is a static analysis tool for Smart Contracts.
- Remix also performs static analysis to your code and is able to spot bugs such as uninitialized storage pointers and reentrancy.
- Oyente is another recently announced analysis tool for Smart Contracts
- Hydra is a “framework for cryptoeconomic contract security, decentralized security bounties”
- Porosity is a “decompiler for Blockchain-based Ethereum Smart-Contracts”
- Manticore is a dynamic binary analysis tool with EVM support
- Ethersplay is a Binary Ninja plugin for EVM

AUTHOR: AKASH KHOSLA
SOURCE

OPENZEPPELIN

What is OpenZeppellin?

- Zeppelin is a library for writing secure smart contracts on Ethereum.

What kind of contracts are available?

- Ownership: declaring, claiming transferring
- Safe Math: takes care of integer overflows and overdrawing
- Push and Pull: deposit and withdraw methods you can trust
- Standard and Basic Tokens: ERC20 implementable and ready out of the box
- GitHub: https://github.com/OpenZeppelin/zeppelin-solidity
- Article: https://blog.zeppelin.solutions/onward-with-ethereum-smart-contract-security-97a827e47702

AUTHOR: COLLIN CHIN

http://zeppelin-solidity.readthedocs.io/en/latest/index.html

SOLGRAPH

What is SolGraph?

- Generates a DOT graph that visualizes function control flow of a Solidity contract and highlights potential security vulnerabilities.

How do I run it / how does it work?

- Github: https://github.com/raineorshine/solgraph

4 HISTORY

WHAT HAPPENED TO THE DAO?

HOW WAS THE DAO HACKED

- **DAO** = Decentralized Autonomous Organization
  - Raise Ether, give tokens that allow you to vote on what projects to fund
- **A total of 3.6m Ether** (worth around $70M at the time) was drained by the hacker. The attack happened due to an exploit found in the splitting function (used to get away from people you don’t agree with in a DAO)
- The attackers withdrew Ether from The DAO smart contract multiple times using the same DAO Tokens
- This was possible due to what is known as a reentrancy exploit

WHAT HAPPENED TO THE DAO?

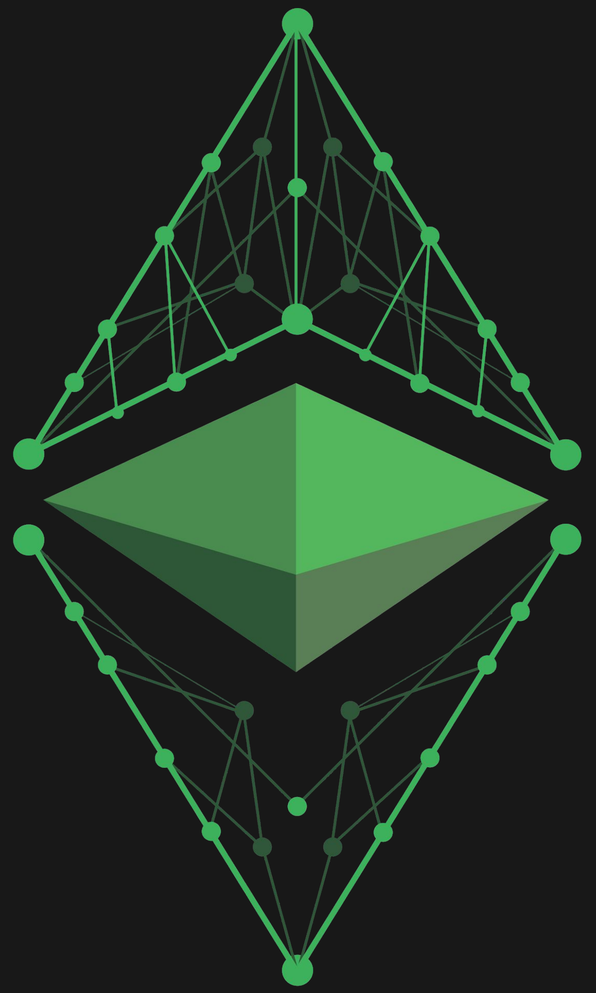
HOW WAS THE DAO HACKED

- In this exploit, the attacker was able to "ask" the smart contract (DAO) to give the Ether back multiple times before the smart contract could update its own balance.
- There were two main issues that made this possible:
  - A recursive attack on `splitDAO` from `withdrawRewardFor`, allowing you to withdraw ~30X as many DAO tokens as you should have access to, but only once
  - A reentrant but non-recursive attack on `splitDAO` from `withdrawRewardFor` through `transfer`, allowing you to double your DAO tokens with every split

The attack was a combination of (1) and (2), allowing you to withdraw ~30X as many DAO tokens as you should have access to, as many times as you want

Result of the attack - **a hard fork** - the new Ethereum is not the old one.

Since the hard fork, the attacker ended up making off with his ethereum classic. That means he got away with about $67.4 million.



ethereum

classic

In this hard fork, the transactions related to the DAO were reverted and all transactions post DAO were still on chain. It was as if the hacker did not exist.

BLOCKCHAIN AT BERKELEY

WHAT ABOUT THE HACKER?

DO THEY GET AWAY WITH THIS

Rumors heard through the grapevine:

- The hacker shorted The DAO tokens before the attack
  - Thus making money when its price tanked.
- Why did the hacker only take ⅓ of the The DAO's funds?
  - they ran out of gas to continue siphoning funds
  - they heard of plans to fork in the hopes that the fork would not go through

SEC ruled that The DAO may have been in violation of U.S. securities law, but no word on the hacker. He ended up cashing out some it in ETC.

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

- A hacker found a vulnerability in the Parity 1.5 client's multisig wallet contract
  - In Bitcoin, multisig wallets are generally implemented off chain
    - Much more difficult to implement
  - On Ethereum, these are written as smart contracts on chain
    - The simplicity of implementing this is a huge selling point for Ethereum
    - We deploy Wallet contracts and store that money in that contract account

parity

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

- Simple attack that allowed the hacker to take ownership of a victim's wallet with a single transaction (etherscan)
  - There were three victims, all of which were ICO projects:
    - Edgeless Casino, Swarm City and æternity
  - Keep in mind that a lot of people use this wallet!
  - **153,037 Ether was stolen ($30+ million)**
  - White hats saved $78 million worth of tokens plus 377,105+ ETH (around $72 million)
    - Tokens were BAT, ICONOMI and some others
    - So ~$150 million (DAO lost 70m at the time)

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

9 js/src/contracts/snippets/enhanced-wallet.sol — View

```
@@ -104,7 +104,7 @@ contract WalletLibrary is WalletEvents {
104
105     // constructor is given number of sigs required to do protected "onlymanyowners" transactions
106     // as well as the selection of addresses capable of confirming them.
107 -   function initMultiowned(address[] _owners, uint _required) {
108         m_numOwners = _owners.length + 1;
109         m_owners[1] = uint(msg.sender);
110         m_ownerIndex[uint(msg.sender)] = 1;
@@ -198,7 +198,7 @@ contract WalletLibrary is WalletEvents {
198     }
```

```
@@ -104,7 +104,7 @@ contract WalletLibrary is WalletEvents {
104
105     // constructor is given number of sigs required to do protected "onlymanyowners" transactions
106     // as well as the selection of addresses capable of confirming them.
107 +   function initMultiowned(address[] _owners, uint _required) internal {
108         m_numOwners = _owners.length + 1;
109         m_owners[1] = uint(msg.sender);
110         m_ownerIndex[uint(msg.sender)] = 1;
@@ -198,7 +198,7 @@ contract WalletLibrary is WalletEvents {
198     }
```

HERE

AUTHOR: AKASH KHOSLA

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

```
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;
         }
     }
}
```

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

    function Wallet(address _owner) {
        // replace the following line with "_walletLibrary = new WalletLibrary();"
        // if you want to try to exploit this contract in Remix.
        _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);
    }
}
```

AUTHOR: AKASH KHOSLA

BLOCKCHAIN
AT BERKELEY

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

- `DELEGATECALL` opcode lets you call an external contract
  - But it lets you call the contract while preserving the call context of the current contract that you're in
  - By call context, i.e. if we are looking at `msg.sender`, we're looking at the current contract's call context, not the contract that we `delegatecall` to
  - Similar to `bind(this)` in javascript, where you pass in the current call context of the function you're in to another
  - This is a completely reasonable function to have

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

    function Wallet(address _owner) {
        // replace the following line with "_walletLibrary = new WalletLibrary();"
        // if you want to try to exploit this contract in Remix.
        _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);
    }
}
```

AUTHOR: AKASH KHOSLA

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

- Problem 1:
  - `initWallet` did not have a guard modifier
  - It doesn't check if the wallet or not has been initialized
    - So if you could call it again, then we're exposed

```
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;
        }
    }
}
```

AUTHOR: AKASH KHOSLA

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

- Problem 2:
  - ‘Derived’ contracts: In this case the `Wallet` contract is created by the `WalletLibrary`
    - This means the wallet can call the `initWallet` function
    - But because the function was not declared internal, that means any derived contract of `Wallet` can call the `initWallet` function

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

    function Wallet(address _owner) {
        // replace the following line with "_walletLibrary = new WalletLibrary();"
        // if you want to try to exploit this contract in Remix.
        _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);
    }
}
```

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

- Since we use `delegatecall`, if the attacker calls the `initWallet` function, the context of the call would be from the attacker's address

```
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;
         }
     }
}
```

AUTHOR: AKASH KHOSLA

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

- Problem 3:
  - In the wallet contract itself, they are using the fallback function as a catch-all to `delegatecall` functions to the `walletLibrary`
- This created an attack vector into the `walletLibrary` from Wallet - we therefore have a publicly accessible way to `initWallet`

TROUBLE!!

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

    function Wallet(address _owner) {
        // replace the following line with "_walletLibrary = new WalletLibrary();"
        // if you want to try to exploit this contract in Remix.
        _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);
    }
}
```

```
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;
         }
     }
```

AUTHOR: AKASH KHOSLA

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

Walking through the attack:

- When an attacker calls `Wallet.initWallet(attacker)`, Wallet's fallback function is triggered
- Wallet's fallback function then `delegatecalls` WalletLibrary
  - That call data consists of the function selector for the `initWallet(address)` function and the attacker's address.

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

    function Wallet(address _owner) {
        // replace the following line with "_walletLibrary = new WalletLibrary();"
        // if you want to try to exploit this contract in Remix.
        _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);
    }
}
```

```
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;
         }
     }
```

AUTHOR: AKASH KHOSLA

WHAT HAPPENED WITH PARITY?

HOW DID GAV WOOD MISS THIS

- WalletLibrary then receives the call data
  - it finds that its `initWallet` function matches the function selector and runs `initWallet(attacker)` in the context of Wallet, setting Wallet's `owner` variable to `attacker`.
- BOOM! The attacker is now the wallet's owner and can withdraw any funds at her leisure.

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

    function Wallet(address _owner) {
        // replace the following line with "_walletLibrary = new WalletLibrary();"
        // if you want to try to exploit this contract in Remix.
        _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);
    }
}
```

```
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;
         }
     }
```

AUTHOR: AKASH KHOSLA

WHAT HAPPENED WITH PARITY?

CONCLUSIONS

- Writing real smart contracts is hard...
- In reality, the `initWallet` function was more complicated and took more parameters
- Solidity is messy in design - it should encourage certain practices
  - Private functions by default
  - Require guards to all functions
  - Keep contracts simple
- The internet is littered with exploitable code - the difference here is the returns and liquidity market is so fast here just economically makes sense
  - A hacker made 30 million dollars in liquid assets..
  - Need for stronger security audits in the space as well as hefty bug bounties.

FUNDAMENTAL TRADEOFFS

Takeaway: Always ask if simplicity or complexity is better

- **Rigid vs. Upgradeable**
  - Rigid: Simple rigid contracts are easy to maintain and protect
  - Upgradeable: Killable, modifiable can allow older contracts to be upgraded without redeployment, but additional attack vectors
- **Monolithic vs. Modular**
  - Monolithic: Contract storage and method locality
  - Modular: Abstract contacts and interfaces
- **Duplication vs. Reuse**
  - Duplication: roll your own math functions that are cheaper on gas
  - Reuse: use OpenZeppelin math functions that are expensive but maintained

AUTHOR: COLLIN CHIN
https://github.com/ConsenSys/smart-contract-best-practices

BLOCKCHAIN
AT BERKELEY

SEE YOU NEXT TIME

Security Exercises

Defending Against Attacks

Best Practices

Testing Functionality

Push/Pull Paradigms

External Calls