Desarrollo Blockchain Ethereum con Solidity

Módulo 4 - Errores conocidos

Errores conocidos

Warnings

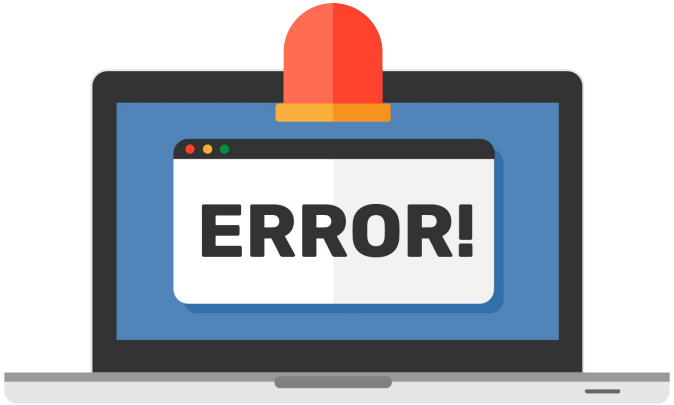
Al desarrollar con Solidity para Ethereum Blockchain, existen criterios básicos a tener en cuenta:

- Es relativamente sospechoso que un contrato se haya compilado con una versión nightly y no con una release. Esta lista no mantiene un registro de las versiones nightly.
- También es sospechoso cuando un contrato fue compilado con una versión que no era la más reciente en el momento que el contrato fue creado. Para contratos creados de otros contratos, tienes que seguir la cadena de creación de vuelta a una transacción y revisar la fecha de esa transacción.
- Es muy sospechoso que un contrato haya sido compilado con un compilador que contiene un error conocido, y que se cree cuando una versión del compilador con una corrección ya haya sido publicada.
- Cada error conocido es público y publicado en GitHub.
- Cada error contiene un conjunto de propiedades que lo identifican y permiten su seguimiento, tales como: *name, summary, description, link, introduced, fixed, publish, severity* y *conditions.*

Errores conocidos

Como en todo sistema, los smart contracts no están exentos de fallos y vulnerabilidades que podrían dejar expuesto a posibles atacantes, mecanismos para poder tomar ”ventaja” del código existente. Entre las principales fallas se encuentran las siguientes:

- Overflow and Underflow.
- Force Ether.
- Keeping Secrets.
- Dangerous DelegateCall.
- Llamada a lo desconocido.
- Reentrancy.



Errores conocidos

Overflow & Underflow

- El sistema de conteo en Solidity se asemeja a un odómetro mecánico: si agrega 1 al valor máximo (256 bits o 2^{256}-1), se volteará y el resultado será 0, un valor que causará un **overflow**.
- Además, si resta 1 de 0 (el número sin signo en Solidity), el resultado retrocederá y será el valor máximo 2^{256}-1, y se producirá un **underflow**.
- Tanto las vulnerabilidades de **Overflow** como las de **Underflow** son bastante peligrosas, pero el caso de **underflow** es más ventajoso para un atacante que busca explotar esta característica.
- Por ejemplo, explotando la vulnerabilidad de underflow, un atacante puede gastar más tokens de los que tiene, su saldo alcanzará el valor máximo y los valores volverán al valor máximo. Por ejemplo, cuando tiene 999 tokens y gasta 1000 tokens, su saldo aumenta a 2^{256}-1 tokens.
- La mejor manera de asegurar un contrato inteligente contra esta vulnerabilidad es usar la biblioteca **SafeMath** de OpenZeppelin.

Errores conocidos

Keeping Secrets

- Esta vulnerabilidad surge cuando un usuario intenta ocultar alguna información en un campo de un contrato inteligente al marcarlo como privado.
- Para hacer esto, el diseñador debe enviar una transacción relevante a los mineros.
- De acuerdo con la transparencia de Blockchain, el contenido de dicha transacción puede ser revisado por cualquier persona interesada y la información en un campo privado puede ser fácilmente revelada.
- Para garantizar el ocultamiento de la información en campos privados, es necesario aplicar tanto técnicas criptográficas como compromisos cronometrados.

Errores conocidos - **Keeping Secrets**

- La vulnerabilidad de guardar secretos a menudo se puede detectar en juegos en línea multijugador sobre la Blockchain.
- Asumamos que hay un contrato inteligente para un juego de probabilidades y emparejamientos con reglas más simples: cada uno de los dos jugadores debe proponer un número y, si la suma de estos números es par, los primeros jugadores ganan y los segundos pierden.

Errores conocidos - **Keeping Secrets**

El contrato inteligente almacena las apuestas de los participantes en un campo privado oculto llamado **players**. Para comenzar un juego, cada jugador debe enviar 1 éter al contrato inteligente. Tan pronto como cada jugador se une al juego, el contrato inteligente especifica el ganador a través del comando **andTheWinner** y le envía 1.8 éter. El 0.2 éter restante se mantiene como tarifa y el propietario lo recopila a través de la función **getProfit**.

Errores conocidos - **Keeping Secrets**

```
contract OddsAndEvens {
struct Player {
address addr;
uint number;
}

Player[2] private players;
uint tot = 0;
address owner;

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

function play(uint _number) payable public {
if (msg.value != 1 ether) revert();
players[tot] = Player(msg.sender, _number);
tot++;
if (tot == 2) andTheWinnerIs();
}

function andTheWinnerIs() private {
uint n = players[0].number
+ players[1].number;
players[n%2].addr.transfer(1800 finney);
delete players;
tot = 0;
}

function getProfite() public {
owner.transfer(address(this).balance);
}
}
```

Errores conocidos

DelegateCall

La estructura de una solicitud de delegación de llamadas es casi la misma que la estructura de una solicitud de llamada. La única diferencia entre los dos es que para la delegación de llamadas, el código de la dirección del destinatario se ejecuta de la misma manera que el código del contrato del llamante.

Entonces, si el argumento de la solicitud de delegación de llamada se establece como **msg.data**, un atacante puede crear el msg.data con tal firma de función para que pueda hacer que el contrato inteligente de la víctima realice una llamada para cualquier función que proporcione.

- Para garantizar el ocultamiento de la información en campos privados, es necesario aplicar tanto técnicas criptográficas como compromisos cronometrados.

Errores conocidos - **DelegateCall**

Este ejemplo es de la vida real, un contrato de Parity que claramente expone vulnerabilidad:

```
contract Wallet {
   address _walletLibrary;
   address owner;


   function Wallet(address _owner) {
      _walletLibrary = <address of pre-deployed WalletLibrary>;
      _walletLibrary.delegatecall(bytes4(sha3("initWallet(address)")), _owner);
   }


   function withdraw(uint amount) returns (bool success) {
      return _walletLibrary.delegatecall(bytes4(sha3("withdraw(uint)")),
amount);
   }


   function () payable {
      _walletLibrary.delegatecall(msg.data);
   }


}
```

Errores conocidos

Llamada a lo desconocido

La vulnerabilidad de “Llamada a lo Desconocido” surge de la función de respuesta del contrato (**receive/fallback**) en Solidity que puede causar daño bajo ciertas condiciones. La función de recepción puede invocarse cuando un llamante de un contrato inteligente envía una solicitud (una acción llamada llamada, delegar llamada, enviar o llamar directa) que invoca ciertas funciones o transfiere ether a otro contrato inteligente.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract LlamadaALoDesconocido {
    address liderDelContrato;
    uint apuestaMaxima;

    receive() external payable {
        require(msg.value > apuestaMaxima);
        require(payable(liderDelContrato).send(apuestaMaxima));
        liderDelContrato = msg.sender;
        apuestaMaxima = msg.value;
    }
}
```

Errores conocidos - **Llamada a lo desconocido**

Para poder explotar la vulnerabilidad, bastó con crear un contrato como el siguiente:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract LlamadaALoDesconocido {
    address liderDelContrato;
    uint apuestaMaxima;

    receive() external payable {
        require(msg.value > apuestaMaxima);
        require(payable(liderDelContrato).send(apuestaMaxima));
        liderDelContrato = msg.sender;
        apuestaMaxima = msg.value;
    }
}

contract Malicia {
  function volverseLider(address _direccion, uint monto) public {
    _direccion.call{ value: monto }("");
  }

  receive() external payable {
    revert();
  }
}
```

Errores conocidos - **Llamada a lo desconocido: Solución**

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract ContratoSolucionado {
    address public liderDelContrato;
    uint public apuestaMaxima;

    mapping (address => uint) transferenciasPendientes;

    constructor() public payable {
        liderDelContrato = msg.sender;
        apuestaMaxima = msg.value;
    }

    function convertirseEnLider() public payable {
        require(msg.value > apuestaMaxima, "Monto insuficiente.");
        transferenciasPendientes[liderDelContrato] += msg.value;
        liderDelContrato = msg.sender;
        apuestaMaxima = msg.value;
    }

    function SaldarDeudas() public {
        uint monto = transferenciasPendientes[msg.sender];
        transferenciasPendientes[msg.sender] = 0;
        msg.sender.transfer(monto);
    }
}
```

Errores conocidos

Reentrancy

- La reentrada es otra vulnerabilidad que precipitó el problema mencionado anteriormente del contrato inteligente de “TheDAO”.
- La vulnerabilidad de reentrada consiste en la estructura de la función de recepción que permite a un atacante invocar repetidamente la función del llamante.
- Como resultado, la vulnerabilidad de reentrada abre la puerta de entrada a la pérdida de gas y fondos almacenados en el contrato inteligente.

Errores conocidos - **Reentrancy**

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Reentrancy {

    mapping(address => uint) balances;

    function Transferir(uint _monto) public {
      require (balances[msg.sender] >= _monto);
      bool boolResult;
      bytes memory bytesResult;
      (boolResult, bytesResult) = msg.sender.call{value:_monto}("");

      if (boolResult) balances[msg.sender] -= _monto;
    }

}
```

Errores conocidos - **Reentrancy: Soluciones**

Existen varias soluciones:

- Una es utilizar un patrón de retiro similar a la solución de Llamada a lo desconocido.
- Otra opción es cambiar el call a send o transfer, ya que limitan la cantidad de gas y hacen que no sea posible el reentrancy.
- Otra opción es el patrón “check effects”, y setear la variable de balance antes que enviar la transacción.
- Otro patrón a utilizar es el “mutex” donde se bloquea la transacción hasta que sea exitosa y se desbloquea. Este patrón es efectivo pero consume más gas.

¡Muchas gracias!

¡Sigamos trabajando!