

Curso desarrollo Blockchain Ethereum con Solidity

Clase 5

Pitfalls

Pitfalls

Como en todo sistema, los smartContracts 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:

- Llamada a lo desconocido
- Reentrancy
- Exception Disorder
- Gasless Send
- Keeping Secrets
- Timestamp Dependence
- Generating Randomness
- Dangerous DelegateCall
- Overflow and Underflow



Pitfalls

Llamada a lo desconocido

La vulnerabilidad de “Llamada a lo Desconocido” surge de la función de respaldo (**fallback function**) en Solidity que puede causar daño bajo ciertas condiciones. La función de respaldo 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.*

```
1  contract TheDAO {
2  mapping (address => uint) public
3  credit;
4
5  function donate(address _to) payable public {
6  credit[_to] += msg.value;
7  }
8
9  function queryCredit(address _to) public view returns (uint) {
10 return credit[_to];
11 }
12
13 function withdraw(uint _amount) public {
14 if (credit[msg.sender] >= _amount) {
15 msg.sender.call.value(_amount)();
16 credit[msg.sender] -= _amount;
17 }
18 }
19
20 function getBalance() public view returns (uint) {
21 return address(this).balance;
22 }
```

Pitfalls

Llamada a lo desconocido

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

```
1  contract TheDAO {
2  mapping (address => uint) public
3  credit;
4
5  function donate(address _to) payable public {
6  credit[_to] += msg.value;
7  }
8
9  function queryCredit(address _to) public view returns (uint) {
10 return credit[_to];
11 }
12
13 function withdraw(uint _amount) public {
14 if (credit[msg.sender] >= _amount) {
15 msg.sender.call.value(_amount)();
16 credit[msg.sender] -= _amount;
17 }
18 }
19
20 function getBalance() public view returns (uint) {
21 return address(this).balance;
22 }
```

Pitfalls

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 respaldo 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.

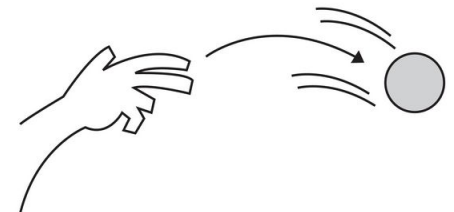
```
...  
// Burn DAO Tokens  
if (balances[msg.sender] == 0)  
    throw;  
withdrawRewardFor(msg.sender);  
totalSupply -= balances[msg.sender];  
balances[msg.sender] = 0;  
paidOut[msg.sender] = 0;  
return true;
```

Pitfalls

Exception Disorder

- El Desorden de excepción es una vulnerabilidad que surge en los contratos inteligentes basados en Solidity bajo varias condiciones:
 - Cuando hay escasez de GAS
 - Cuando se encuentra con un límite de pila de llamadas
 - Durante la ejecución de un comando **throw**
- La manera en que Solidity se comporta con excepciones a menudo difiere y esta es la causa de la vulnerabilidad de **exception disorder**.
- Debido a esta vulnerabilidad, Solidity no comprueba los errores de envío o las solicitudes de llamadas y esto lleva a la reversión de una transacción y la **pérdida de gas**. Por ejemplo, supongamos que hay una transacción que consiste en una secuencia de varias llamadas.
- La vulnerabilidad del trastorno de excepción puede abrir una ventana para el comportamiento malicioso, como lo fue en el caso del esquema de Ponzi de GovernMental

throw



Pitfalls

Gasless Send

- La causa de la vulnerabilidad del “envío sin gas” es la excepción de falta de gas.
- Esta excepción aparece cuando se transfiere el Ether a través de la función de envío a un destinatario con una función de respaldo costosa limitada a un máximo de 2300 unidades de gas. En los casos en que la excepción de falta de gas no se maneja adecuadamente, los usuarios malintencionados pueden fingir que ya han enviado Ether al destinatario al tiempo que retienen la cantidad de envío en una Wallet.
- El mejor ejemplo de la vulnerabilidad del envío sin gas es el juego “King of the Ether Throne”.
- El objetivo de este juego es que los usuarios envíen una cierta cantidad de éter al contrato inteligente de KotET para convertirse en un nuevo “Rey”.
- Una parte de los fondos enviados se destina al contrato inteligente y otra parte se envía al rey anterior como compensación por la destitución.
- El propietario del juego KotET logró diseñar el contrato inteligente de tal manera que la compensación no se envió al rey anterior, sino que fue conservada por el propietario del juego.
- <https://www.kingoftheether.com>



Pitfalls

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.
- 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.



Pitfalls

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**.

```
1  contract OddsAndEvens {
2      struct Player {
3          address addr;
4          uint number;
5      }
6
7      Player[2] private players;
8      uint tot = 0;
9      address owner;
10
11     constructor () public {
12         owner = msg.sender;
13     }
14
15     function play(uint _number) payable public {
16         if (msg.value != 1 ether) revert();
17         players[tot] = Player(msg.sender, _number);
18         tot++;
19         if (tot == 2) andTheWinnerIs();
20     }
```

```
21
22     function andTheWinnerIs() private {
23         uint n = players[0].number
24         + players[1].number;
25         players[n%2].addr.transfer(1800 finney);
26         delete players;
27         tot = 0;
28     }
29
30     function getProfite() public {
31         owner.transfer(address(this).balance);
32     }
33 }
```

Pitfalls

Timestamp Dependence

- Para realizar cualquier operación en la Blockchain, por ejemplo, para transferir Ether, un contrato inteligente recibe un **timestamp** que especifica la hora en que se generó el bloque.
- Si un minero malicioso tiene interés en ejecutar un contrato inteligente para algún ataque o estafa, puede manipular el timestamp para el bloque generado adecuado para su propio propósito.
- Esta vulnerabilidad de dependencia de timestamp fue explotada por los propietarios de la pirámide de **GovernMental**.
- El minero malicioso generó un bloqueo para su transacción con el timestamp modificado que retrasó su transacción para ser la última y de esta manera ganó los fondos del contrato inteligente.

	_time ↕	IndexTime ↕	diff ↕
1	6/19/11 5:59:59.336 PM	06/19/2011 18:00:02	2.664
2	6/19/11 5:59:59.334 PM	06/19/2011 18:00:00	0.666
3	6/19/11 5:59:59.331 PM	06/19/2011 18:00:00	0.669
4	6/19/11 5:59:59.323 PM	06/19/2011 18:00:02	2.677
5	6/19/11 5:59:59.187 PM	06/19/2011 18:00:00	0.813
6	6/19/11 5:59:59.187 PM	06/19/2011 18:00:00	0.813
7	6/19/11 5:59:59.187 PM	06/19/2011 18:00:00	0.813
8	6/19/11 5:59:59.187 PM	06/19/2011 18:00:00	0.813
9	6/19/11 5:59:59.187 PM	06/19/2011 18:00:00	0.813
10	6/19/11 5:59:59.187 PM	06/19/2011 18:00:00	0.813

« prev 1 2 3 4 5 6 7 8 9 10 next »

Pitfalls

Generating Randomness

- Muchos contratos inteligentes, como los realizados para crear loterías o juegos en línea, generan números aleatorios para mantener su actividad.
- Para generar un número aleatorio, se requiere enviar una transacción apropiada a la Blockchain.
- Un minero malicioso puede explotar este potencial para organizar el bloque generado con una transacción particular que coincida con el resultado del número generado al azar y usar esta técnica para sus propios fines.



Pitfalls

Dangerous 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.

```
1  contract Wallet{
2      function() payable { //fallback function
3          if (msg.value > 0)
4              Deposit(msg.sender, msg.value);
5          else if (msg.data.length > 0)
6              _walletLibrary.delegatecall(msg.data);
7      }
8  }
9  contract WalletLibrary {
10     function initWallet(address[] _owners, uint _required, uint _daylimit) {
11         initDaylimit(_daylimit);
12         initMultiowned(_owners, _required);
13     }
14 }
```

Pitfalls

Overflow and Underflow

- El sistema de conteo en Solidity se asemeja a un odómetro mecánico: si agrega 1 al valor máximo ($2^{256}-1$), se voltará 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



Limites de GAS & Bucles



Limites de GAS & Bucles

- Los bucles que no tienen un número fijo de iteraciones, por ejemplo, bucles que dependen de valores de almacenamiento, tienen que usarse **con cuidado**
- Debido al límite de gas del bloque, las transacciones sólo pueden consumir una cierta cantidad de gas
- Ya sea explícitamente, o por una operación normal, el número de iteraciones en un bucle puede crecer más allá del límite de gas, lo que puede causar que el **contrato se detenga** por completo en un cierto punto
- Esto no se aplica a funciones **constant** que sólo se llaman para leer información de la blockchain*
- Es altamente recomendable ser siempre explícito con estos casos en la documentación de nuestros contratos



Limites de GAS & Bucles

Es posible averiguar el precio actual del GAS via etherscan
<https://etherscan.io/block/{blockNumber}>

Block Information  	
Height:	6771334
TimeStamp:	2 mins ago (Nov-25-2018 05:33:47 PM +UTC)
Transactions:	25 transactions and 10 contract Internal Transactions in this Block
Hash:	0xbf89cc3e7e6c8a81e8ec3f5354630ae8946e52f41de17f9612c88adcc66fc388
Parent Hash:	0x3442d3b8f31fc24dabbf71477f547b50f0061e0155eb06d6b7ea6fd8fa56eb06
Sha3Uncles:	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
Mined By:	0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5 (Nanopool) in 7 secs
Difficulty:	2,703,649,853,347,188
Total Difficulty:	8,001,866,980,950,282,516,744
Size:	6449 bytes
Gas Used:	1,745,638 (21.82%)
Gas Limit:	8,000,029
Nonce:	0x5a70c660069c1a11
Block Reward:	3.016779409267406954 Ether (3 + 0.016779409267406954)
Uncles Reward:	0
Extra Data:	nanopool.org (Hex:0x6e616e6f706f6f6c2e6f7267)

Warnings

Warnings

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

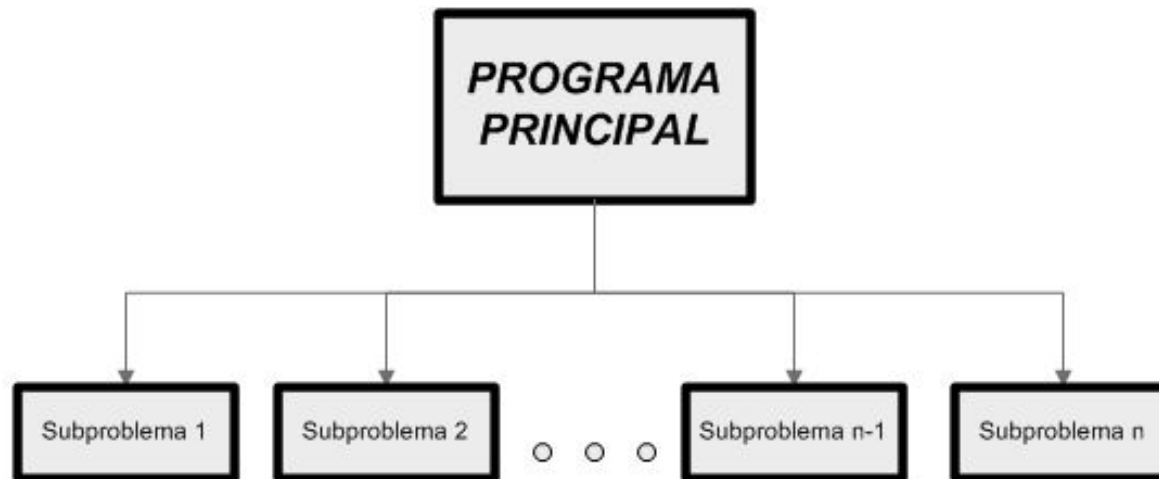
- Es relativamente sospechoso que un contrato se haya compilado con una versión nightly y no con una release. Esta lista no mantiene un registro de las versiones nightly.
- También es sospechoso cuando un contrato fue compilado con una versión que no era la más reciente en el momento que el contrato 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



Modularización

Modularización

- Mantener tus contratos **pequeños** y **fáciles** de entender
- Separar las funcionalidades no relacionadas en otros contratos o en librerías
- Limitar la cantidad de variables locales
- Limitar la longitud de las funciones
- Documentar las funciones para que otros puedan ver cuál era la intención del código y para ver si hace algo diferente de lo que pretendía



ABI

ABI

- La **Application Binary Interface** (Interfaz Binaria de Aplicación) o ABI es el modo estándar de interactuar con contratos en el ecosistema Ethereum, tanto desde fuera de la blockchain como en interacciones contrato-contrato
- Los datos se codifican siguiendo su tipo acorde a esta especificación
- La ABI está **fuertemente tipada**, es conocida en tiempo de compilación y es estática
- Los contratos tendrán las definiciones de la interfaz de cada contrato que vayan a llamar en tiempo de compilación
- Esta especificación no abarca los contratos cuya interfaz sea dinámica o conocida exclusivamente en tiempo de ejecución. Estos casos, de volverse importantes, podrían manejarse adecuadamente como servicios contruidos dentro del ecosistema Ethereum.



Desarrollando "a prueba de fallos"

Desarrollando "a prueba de fallos"

- Aunque hacer que tu sistema sea completamente descentralizado eliminará cualquier intermediario, puede que sea una buena idea, especialmente para nuevo código, incluir un sistema a prueba de fallos
- Puedes agregar una función a tu contrato que realice algunas comprobaciones internas como "¿Se ha filtrado Ether?", "¿La sumatoria de los tokens es igual al account balance?", etc
- Recordad que no se puede usar mucho gas para eso, así que ayuda mediante computaciones **off-chain*** podrían ser necesarias.
- Si los chequeos fallan, el contrato automáticamente cambia a modo a prueba de fallos, donde, por ejemplo, se desactivan muchas funciones, da el control a una entidad tercera de confianza o se convierte en un contrato del tipo "devolveme mi dinero"



Patrones comunes

Patrones comunes

Al desarrollar con Solidity para la Ethereum Blockchain encontraremos varios patrones sumamente útiles para garantizar tanto la correcta ejecución de nuestros contratos, como la seguridad del dinero (Ether) asociado.

Entre estos patrones encontramos los siguientes:

- Retiro desde Contratos
- Máquina de estados



Patrones comunes

Retiro desde Contratos

- El método recomendado para enviar fondos después de una ejecución es usar el patrón de retiro.
- Aunque el método más intuitivo de enviar Ether, como resultado de una acción, es una llamada de envío directo, no se recomienda, ya que presenta un riesgo potencial para la seguridad
- Siempre es recomendable leer estas y otras consideraciones de seguridad desde el sitio oficial
- Un ejemplo del patrón de retiro en la práctica en un contrato donde el objetivo es enviar la mayor cantidad de dinero al contrato para convertirse en el "más rico" es el caso de "King of the Ether"*



Patrones comunes

Analicemos el siguiente contrato inteligente

```
pragma solidity ^0.4.11;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    function WithdrawalContract() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable returns (bool) {
        if (msg.value > mostSent) {
            pendingWithdrawals[richest] += msg.value;
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        msg.sender.transfer(amount);
    }
}
```

Patrones comunes

- Ahora, con un patrón de envío más intuitivo

```
pragma solidity ^0.4.11;

contract SendContract {
    address public richest;
    uint public mostSent;

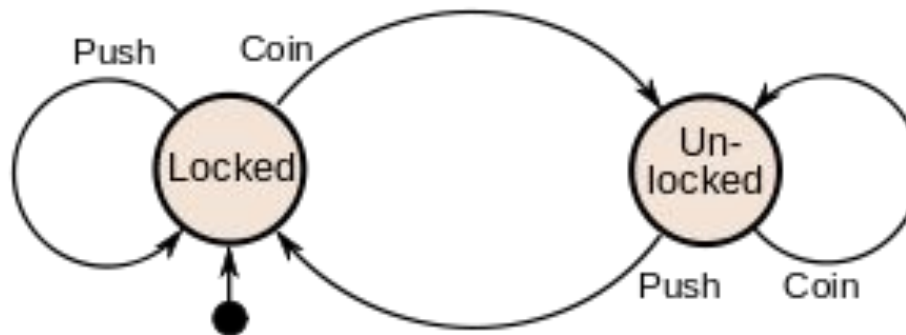
    function SendContract() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable returns (bool) {
        if (msg.value > mostSent) {
            // This line can cause problems (explained below).
            richest.transfer(msg.value);
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }
}
```


Patrones comunes

Máquina de estados

- Los contratos a menudo actúan como una máquina de estados, lo que significa que tienen ciertas etapas en donde se comportan de manera diferente o en donde distintas funciones pueden ser llamadas.
- Una llamada de función a menudo termina una etapa y pasa el contrato a la siguiente etapa (especialmente si el contrato modela la interacción).
- También es común que algunas etapas se alcancen automáticamente en cierto punto en el tiempo.
- Un ejemplo de esto es el contrato de subastas a ciegas que comienza en la etapa "aceptando pujas a ciegas", luego pasa a "revelando pujas" que es finalizado por "determinar resultado de la subasta".
- Los modificadores de funciones se pueden usar en esta situación para modelar los estados y evitar el uso incorrecto del contrato.



Restringiendo el acceso

Restringiendo el acceso

Restricción de acceso

- La restricción de acceso es un patrón conocido (y muy utilizado) en los SmartContracts.
- No será posible restringir a absolutamente nadie la lectura de la transacción y la información asociada a la misma
- Es posible hacer que este proceso sea más difícil, usando encriptación, pero si el contrato está diseñado para leer información también lo podrán hacer otras personas
- Es posible restringir el acceso de lectura a tus contratos.
- Esto funciona así por default a menos que declaremos las variables como públicas
- Además, es posible restringir quien puede realizar modificaciones de estado dentro de nuestros contratos o incluso quien puede llamar a determinadas funciones de los contratos



Restringiendo el acceso

La utilización de modificadores de función hace posibles este tipo de restricciones de manera sumamente sencilla

```
// A certain address
modifier onlyBy(address _account)
{
    require(
        msg.sender == _account,
        "Sender not authorized."
    );
    // Do not forget the "_";! It will
    // be replaced by the actual function
    // body when the modifier is used.
    _;
}

/// Make `_newOwner` the new owner of this
/// contract.
function changeOwner(address _newOwner)
    public
    onlyBy(owner)
{
    owner = _newOwner;
}

modifier onlyAfter(uint _time) {
    require(
        now >= _time,
        "Function called too early."
    );
    _;
}
```

Consideraciones de Overflow

Consideraciones de overflow

- Un desbordamiento de búfer (**overflow**) es un error de software que se produce cuando un programa no controla adecuadamente la cantidad de datos que se copian sobre un área de memoria reservada a tal efecto (**buffer**)
- Solidity no está exento de este tipo de fallos

Imaginemos que tenemos un uint8, que solo puede tener 8 bits (Lo que significa que puede almacenar hasta el numero 255)

```
uint8 number = 255;  
number++;
```

El resultado de esa operación es que ahora la variable number tiene un valor igual a cero

Esto es debido a que al sumarle 1 al número 255 (o 11111111 en binario), éste se resetea

En contracara, si el numero fuese cero y se le restase uno, éste se convertiría en 255 (caso de underflow)*