以下是对链上资金转入与撤销代码实例

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.8;

import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

#从chainlink上导入AggregatorV3接口合约

import "./PriceConverter.sol";

这个合约为定义的价格转换合约 因类似于库的使用在此简单的不予展示

error NotOwner();

检测是否为本管理者

contract FundMe {

using PriceConverter for uint256;

mapping(address => uint256) public addressToAmountFunded;

address[] public funders;

address public /\* immutable \*/ i\_owner;

定义 immutable后在函数调用中不可以改变函数值

uint256 public constant MINIMUM\_USD = 50 \* 10 \*\* 18;

constructor() {

i\_owner = msg.sender;

}

function fund() public payable {

定义关键字payable才能交易

require(msg.value.getConversionRate() >= MINIMUM\_USD, "You need to spend more ETH!");

// require(PriceConverter.getConversionRate(msg.value) >= MINIMUM\_USD, "You need to spend more ETH!");

addressToAmountFunded[msg.sender] += msg.value;

funders.push(msg.sender);

}

function getVersion() public view returns (uint256){

// ETH/USD price feed address of Goerli Network.

AggregatorV3Interface priceFeed = AggregatorV3Interface(0xD4a33860578De61DBAbDc8BFdb98FD742fA7028e);

return priceFeed.version();

}

modifier onlyOwner {

修改关键字 定义之后下面使用函数时带有onlyowner关键字的函数才能被管理者调用

// require(msg.sender == owner);

if (msg.sender != i\_owner) revert NotOwner();

\_;

}

function withdraw() public onlyOwner {

定义了一个撤销资金的函数

for (uint256 funderIndex=0; funderIndex < funders.length; funderIndex++){

address funder = funders[funderIndex];

addressToAmountFunded[funder] = 0;

}

funders = new address[](0);

// // transfer

// payable(msg.sender).transfer(address(this).balance);

// // send

// bool sendSuccess = payable(msg.sender).send(address(this).balance);

// require(sendSuccess, "Send failed");

// call

(bool callSuccess, ) = payable(msg.sender).call{value: address(this).balance}("");

require(callSuccess, "Call failed");

}

// Explainer from: https://solidity-by-example.org/fallback/

// Ether is sent to contract

// is msg.data empty?

// / \

// yes no

// / \

// receive()? fallback()

// / \

// yes no

// / \

//receive() fallback()

fallback() external payable {

fund();

}

receive() external payable {

fund();

}

}

上面演示了solidity在遇到接受函数和回滚函数时的情况 于是我们在上面分别定义了两个函数用来实现防止其他用户错误使用fundme函数时的回滚情况；

以下是学习solidity语言过程中的一些相似疑难笔记；

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | 1. **关键词** |  1. return 2. storage/memory   /calldata   1. mapping（映射） 2. constant/immutable 3. 插入排序 4. event（事件） 5. inheritance（继承） 6. 抽象合约(abstract)和接口（interface） 7. 抛出异常   （Error/Require/Assert）   1. 函数重载（overloading) 2. receive和fallback 3. ETH3种发送方法（transfer/send/call) 4. 调用合约（call） 5. 低级函数(Delegatecall/call) | |  | | --- | | 1. **笔记栏** |  1. returns加在函数名后面，用于声明返回的变量类型及变量名；return用于函数主体中，返回指定的变量。 2. 合约里的状态变量默认都是storage，存储在链上; 函数里的参数和临时变量一般用memory，存储在内存中，不上链; calldata和memory类似，存储在内存中，不上链。与memory的不同点在于calldata变量不能修改（immutable），一般用于函数的参数。 3. 声明映射的格式为mapping(\_KeyType => \_ValueType) 4. constant变量必须在声明的时候初始化，之后再也不能改变。尝试改变的话，编译不通过。immutable变量可以在声明时或构造函数中初始化，因此更加灵活。 5. solidity中最常用的变量类型是uint，也就是正整数，取到负值的话，会报underflow错误。 6. event特点：响应：应用程序（[ethers.js](https://learnblockchain.cn/docs/ethers.js/api-contract.html#id18)）可以通过RPC接口订阅和监听这些事件，并在前端做响应。经济：事件是EVM上比较经济的存储数据的方式，每个大概消耗2,000 gas；相比之下，链上存储一个新变量至少需要20,000 gas。 7. inheritance（继承）：包括简单继承，多重继承，以及修饰器（modifier，也可用overrid重写）和构造函（constructor）的继承。 8. 某个函数缺少主体{}中的内容，则必须将该合约标为abstract,对未实现的函数需要加virtual，以便子合约重写。接口是智能合约的骨架，定义了合约的功能以及如何触发它们（比如ERC20和ERC721）。 9. 一般error方法gas最少，其次是assert，require方法消耗gas最多！因此，error既可以告知用户抛出异常的原因，又能省gas，要多用。 10. 允许函数进行重载,名字相同但输入参数类型不同的函数可以同时存在，他们被视为不同的函数;但是不允许修饰器（modifier）重载 11. 合约接收ETH时，触发的规则如下：msg.data为空且存在receive()时，会触发receive()；msg.data不为空或不存在receive()时，会触发fallback()，此时fallback()必须为payable。 12. call没有gas限制，最为灵活，是最提倡的方法；transfer有2300gas限制，但是发送失败会自动revert交易，是次优选择；send有2300gas限制，而且发送失败不会自动revert交易，几乎没有人用它。 13. call虽然不是调用合约的推荐方法（因为不安全），但是他能让我们在不知道源代码和ABI的情况下调用目标合约，很有用。 14. 当A合约通过B合约调用C合约时，BcallC作用于C的语境，即合约C的状态变量被改变了；而delegatecall与call不同是BdelegatecallC作用于B的语境，合约B的状态变量被改变了，合约C没有改变。另外目前delegatecall最大的应用是代理合约和EIP-2535 Diamonds（钻石）。   call：C的num由0改变为10；delegatecall：B的num被改变为10。 |

|  |
| --- |
| **总结栏**  以上是我在学习solidity语言中总结的一些学习笔记，总结性比较强，但方便梳理记忆。细节在我学习的网站https://github.com/AmazingAng/WTF-Solidity。 |

**合约溢出漏洞**

# 1.溢出事件

## BEC合约

**2018年4月22日，黑客对BEC智能合约发起攻击，凭空取出：**

**57,896,044,618,658,100,000,000,000,000,000,000,000,000,000,000,000,000,000,000.792003956564819968个BEC代币并在市场上进行抛**

**售，BEC随即急剧贬值，价值几乎为0，该市场瞬间土崩瓦解。**

* 1. **SMT项目**

**2018年4月25日，SMT项目方发现其交易存在异常，黑客利用其函数漏洞创造了：**

**65,133,050,195,990,400,000,000,000,000,000,000,000,000,000,000,000,000,000,000+50,659,039,041,325,800,000,000,000,000,000,000,000,000,00**

**的SMT币，火币Pro随即暂停了所有币种的充值提取业务。**

* 1. **FNT合约**

**2018年12月27日，以太坊智能合约Fountain(FNT)出现整数溢出漏洞，黑客利用其函数漏洞创造了：**

**2+115792089237316195423570985008687907853269984665640564039457584007913129639935的SMT币。**

**2.整数溢出简介**

* 1. **整数溢出原理**

**由于计算机底层是二进制，任何十进制数字都会被编码到二进制。溢出会丢弃最高位，导致数值不正确。如：八位无符号整数类型的最大值是255，翻译到二进制是1111 1111；当再加一时，当前所有的1都会变成0，并向上进位。但由于该整数类型所能容纳的位置已经全部是1了，再向上进位，最高位会被丢弃，于是二进制就变成了0000 0000。（注：有符号的整数类型，其二进制最高位代表正负。所以该类型的正数溢出会变成负数，而不是**

**零。）**

* 1. **solidity中的整数溢出**

**在solidity中，对于0.8.0以下的版本，也存在整数溢出问题。解决办法就是使用SafeMath库来修补溢出漏洞，源码如下：**

library SafeMath {

function mul(uint256 a, uint256 b) internal constant returns (uint256) {

uint256 c = a \* b;

assert(a == 0 || c / a == b);

return c;

}

function div(uint256 a, uint256 b) internal constant returns (uint256) {

uint256 c = a / b;

return c;

}

function sub(uint256 a, uint256 b) internal constant returns (uint256) {

assert(b <= a);

return a - b;

}

function add(uint256 a, uint256 b) internal constant returns (uint256) {

uint256 c = a + b;

assert(c >= a);

return c;

}

}

**solidity 0.8以上的版本发生运行时溢出会直接revert，修复的方式就是不允许溢出。**

**3.攻击模拟**

* 1. **源码如下：**

contract ZT is ERC20("ZERO TOKEN", "ZT"){

//RULE 1

bytes32 constant RULE\_WITHDRAW\_WANT = keccak256(abi.encodePacked("withdraw"));

//RULE 2

bytes32 constant RULE\_NONE\_WANT = keccak256(abi.encodePacked("depositByValue"));

constructor()public{

\_mint(msg.sender,10000000\*10\*\*18);

}

function depositByWant(uint \_amount)external payable{

uint amount = \_amount.mul(10\*\*18);

require(msg.value>=amount,"you want to trick me?");

MkaheChange(msg.sender,amount,RULE\_NONE\_WANT);

}

function withdraw(uint \_amount)external payable returns(bool){

uint amount = \_amount.mul(10\*\*18);

require(balanceOf(msg.sender)>=amount);

\_balances[msg.sender] = \_balances[msg.sender].sub(amount);

return MkaheChange(msg.sender,amount,RULE\_WITHDRAW\_WANT);

}

function MkaheChange(address to,uint amount,bytes32 ID)internal returns(bool){

if(ID==RULE\_NONE\_WANT)

{

\_balances[msg.sender]=\_balances[msg.sender].add(amount);

return true;

}else if(ID==RULE\_WITHDRAW\_WANT){

bool a;

(a,)=payable(to).call.value(amount)("");

require(a,"withdraw fail");

return true;

}

else{

return false;

}

}

fallback()external payable{

MkaheChange(

msg.sender,

msg.value,

RULE\_NONE\_WANT

);

}

}

contract ZTstakepool{

ZT token;

uint totalsupply;

string symbol;

mapping(address=>uint)internal workbalance;

mapping(address=>bool)internal passed;

struct userInfo{

uint amount;

uint duration;

uint startTime;

}

mapping(address=>userInfo)internal userDetails;

constructor()public{

token =new ZT();

symbol = "stTGT";

totalsupply = token.balanceOf(address(this));

}

function getDetails(address account)public view returns(userInfo memory){

return userDetails[account];

}

function workBalanceOf(address account)public view returns(uint){

bool pass=passed[account];

if(pass){

return workbalance[account];

}else{

return 0;

}

}

function Zt()public view returns(address){

return address(token);

}

function stake(uint amount,uint blocknumber)external{

require(blocknumber>=1,"At least 1 block");

userInfo storage user = userDetails[msg.sender];

user.startTime = block.number;

user.duration = blocknumber;

user.amount += amount;

token.transferFrom(msg.sender,address(this),amount\*10\*\*18);

workbalance[msg.sender] += blocknumber;

}

function unstake()external{

userInfo storage user = userDetails[msg.sender];

require(block.number>=user.startTime+user.duration,"you are in a hurry ");

passed[msg.sender] = true;

uint amount = user.amount;

user.amount = 0;

token.transfer(msg.sender,amount\*10\*\*18);

}

function swap(address from,address to,uint amount)external{

require(from==address(this)&&to==address(token));

uint balance = workBalanceOf(msg.sender);

require(balance>=amount,"exceed");

workbalance[msg.sender] -= amount;

token.transfer(msg.sender,amount\*10\*\*18);

}

}

contract setup{

ZTstakepool public stakePool;

ZT public erc20;

bool solve;

constructor()public{

stakePool =new ZTstakepool();

erc20 = ZT(payable(stakePool.Zt()));

}

function isSolved()public view returns(bool){

return solve;

}

function complete()public{

require(erc20.balanceOf(msg.sender)>=500000\*10\*\*18);

solve = true;

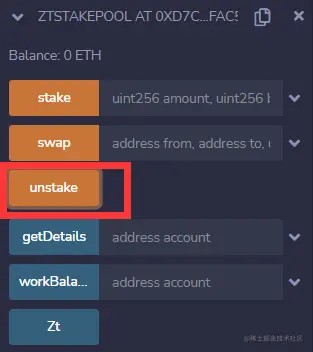
}

}

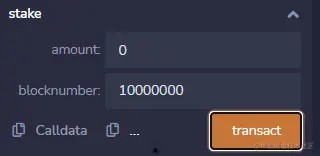
## 合约漏洞查找

**在源码中我们可以看到，其它两个合约由合约setup生成，所以我们先部署setup合约，通过观察可以发现要想isSolved函数返回true，需要满足erc20.balanceOf(msg.sender)>=500000\*10\*\*18，即产生溢出，因此攻击此合约只需要调用参数满足溢出条件即可，我们在上面的代码中可以看到，溢出可能发生在swap函数中。**

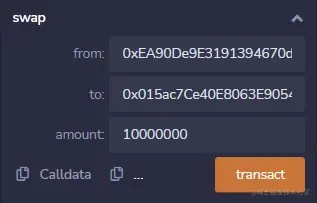
* 1. **开始攻击**

**先执行unstake函数使得passed[msg.sender] = true**

**然后我们就可以执行stake函数了（注意：此时因为我们需要溢出，所以数值应该尽可能大以满足溢出条件）**

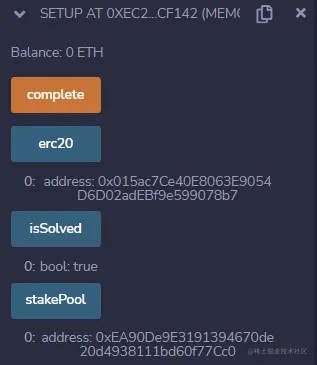


**此时我们调用swap函数提取出ZT**



**此时我们查看ZT合约里的balance函数，可以看到已经产生了溢出**

**调用complete和isSolved完成攻击获得flag**



**获得flag**

