Segurança de contrato inteligente

A segurança é uma das considerações mais importantes ao escrever contratos inteligentes. No campo da programação de contratos inteligentes, os erros são caros e facilmente explorados. Neste capítulo, veremos as melhores práticas de segurança e padrões de design, bem como "antipadrões de segurança", que são práticas e padrões que podem introduzir vulnerabilidades em nossos contratos inteligentes.

Assim como em outros programas, um contrato inteligente executará exatamente o que está escrito, o que nem sempre é o que o programador pretendia. Além disso, todos os contratos inteligentes são públicos e qualquer usuário pode interagir com eles simplesmente criando uma transação. Qualquer vulnerabilidade pode ser explorada e as perdas são quase sempre impossíveis de serem recuperadas. Portanto, é fundamental seguir as melhores práticas e usar padrões de design bem testados.

Práticas recomendadas de segurança

*A programação defensiva* é um estilo de programação particularmente adequado para contratos inteligentes. Ele enfatiza o seguinte, todos os quais são práticas recomendadas:

**Minimalismo/simplicidade**

A complexidade é inimiga da segurança. Quanto mais simples o código, e quanto menos ele faz, menores são as chances de ocorrer um bug ou efeito imprevisto. Ao se envolver pela primeira vez na programação de contrato inteligente, os desenvolvedores geralmente são tentados a tentar escrever muito código. Em vez disso, você deve examinar seu código de contrato inteligente e tentar encontrar maneiras de fazer menos, com menos linhas de código, menos complexidade e menos "recursos". Se alguém lhe disser que seu projeto produziu "milhares de linhas de código" para seus contratos inteligentes, você deve questionar a segurança desse projeto. Mais simples é mais seguro.

**Reutilização de código**

Tente não reinventar a roda. Se já existir uma biblioteca ou contrato que faça a maior parte do que você precisa, reutilize-a. Dentro do seu próprio código, siga o princípio DRY: Don't Repeat Yourself. Se você vir algum trecho de código repetido mais de uma vez, pergunte-se se ele pode ser escrito como uma função ou biblioteca e reutilizado. O código que foi amplamente usado e testado é provavelmente mais seguro do que qualquer novo código que você escreve. Cuidado com a síndrome “Não inventado aqui”, onde você é tentado a “melhorar” um recurso ou componente construindo-o do zero. O risco de segurança é muitas vezes maior do que o valor da melhoria.

**Qualidade do código**

O código de contrato inteligente é implacável. Cada bug pode levar à perda monetária. Você não deve tratar a programação de contrato inteligente da mesma forma que a programação de uso geral. Escrever DApps no Solidity não é como criar um widget da web em JavaScript. Em vez disso, você deve aplicar metodologias rigorosas de engenharia e desenvolvimento de software, como faria na engenharia aeroespacial ou em qualquer outra disciplina igualmente implacável. Depois de "lançar" seu código, há pouco que você possa fazer para corrigir quaisquer problemas.

**Legibilidade/auditabilidade**

Seu código deve ser claro e fácil de compreender. Quanto mais fácil for ler, mais fácil será auditar. Os contratos inteligentes são públicos, pois todos podem ler o bytecode e qualquer um pode fazer engenharia reversa. Portanto, é benéfico desenvolver seu trabalho em público, usando metodologias colaborativas e de código aberto , para aproveitar a sabedoria coletiva da comunidade de desenvolvedores e se beneficiar do mais alto denominador comum de desenvolvimento de código aberto. Você deve escrever um código bem documentado e fácil de ler, seguindo o estilo e as convenções de nomenclatura que fazem parte da comunidade Ethereum.

**Cobertura de teste**

Teste tudo o que puder. Os contratos inteligentes são executados em um ambiente de execução pública, onde qualquer pessoa pode executá-los com qualquer entrada que desejar. Você nunca deve supor que a entrada, como argumentos de função, seja bem formada, adequadamente delimitada ou tenha um propósito benigno. Teste todos os argumentos para garantir que estejam dentro dos intervalos esperados e formatados corretamente antes de permitir que a execução do seu código continue.

Riscos de segurança e antipadrões

Como programador de contratos inteligentes, você deve estar familiarizado com os riscos de segurança mais comuns, para poder detectar e evitar os padrões de programação que deixam seus contratos expostos a esses riscos. Nas próximas seções, veremos diferentes riscos de segurança, exemplos de como as vulnerabilidades podem surgir e contramedidas ou soluções preventivas que podem ser usadas para resolvê-las.

Reentrada

Um dos recursos dos contratos inteligentes Ethereum é a capacidade de chamar e utilizar o código de outros contratos externos. Os contratos também geralmente lidam com ether e, como tal, geralmente enviam ether para vários endereços de usuários externos. Essas operações exigem que os contratos apresentem chamadas externas. Essas chamadas externas podem ser sequestradas por invasores, que podem forçar os contratos a executar mais código (por meio de uma função de fallback), incluindo chamadas de volta para si mesmos. Ataques desse tipo foram usados no infame [hack DAO](http://bit.ly/2DamSZT) .

Para ler mais sobre ataques de reentrância , veja o artigo de Gus Guimareas [postagem no blog](http://bit.ly/2zaqSEY) sobre o assunto e as [melhores práticas de contrato inteligente da Ethereum](http://bit.ly/2ERDMxV) .

A vulnerabilidade

Esse tipo de ataque pode ocorrer quando um contrato envia ether para um endereço desconhecido. Um invasor pode construir cuidadosamente um contrato em um endereço externo que contenha código malicioso na função de fallback. Assim, quando um contrato envia ether para este endereço, ele invocará o código malicioso. Normalmente , o código malicioso executa uma função no contrato vulnerável, realizando operações não esperadas pelo desenvolvedor. O termo “ reentrância ” vem do fato de que o contrato malicioso externo chama uma função no contrato vulnerável e o caminho de execução do código “ *reentra* ” nele. Para esclarecer isso, considere o contrato vulnerável simples em [EtherStore.sol](file:///\\\\wsl.localhost\\Ubuntu-20.04\\home\\syr3fx\\Code\\github\\maitriser-ca\\LivreMaitriserEthereum\\adoc\\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html" \l "etherstore_vulnerable) , que atua como um cofre Ethereum que permite que os depositantes retirem apenas 1 ether por semana.

*Exemplo 1. EtherStore.sol*

contrato EtherStore {

uint256 retirada públicaLimit = 1 éter;

mapping( address => uint256) public lastWithdrawTime ;

mapeamento( endereço => uint256) saldos públicos;

função depositFunds ( ) externo a pagar {

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

}

função retireFunds (uint256 \_ weiToWithdraw ) public {

require(balances[ msg.sender ] >= \_ weiToWithdraw );

//limita a retirada

require( \_ weiToWithdraw <= retiradaLimit );

// limita o tempo permitido para retirar

require( now >= lastWithdrawTime [ msg.sender ] + 1 semanas);

require( msg.sender .call.value (\_ weiToWithdraw )());

saldos[ msg.sender ] -= \_ weiToWithdraw ;

lastWithdrawTime [ msg.sender ] = agora;

}

}

Este contrato tem duas funções públicas, depositFunds e retireFunds . A função depositFunds simplesmente incrementa o saldo do remetente. A função retireFunds permite que o remetente especifique a quantidade de wei a ser retirada. Esta função deve ser bem sucedida apenas se o valor solicitado para retirada for inferior a 1 ether e uma retirada não ocorreu na última semana.

A vulnerabilidade está na linha 17, onde o contrato envia ao usuário a quantidade solicitada de ether. Considere um invasor que criou o contrato em [Attack.sol](file:///\\\\wsl.localhost\\Ubuntu-20.04\\home\\syr3fx\\Code\\github\\maitriser-ca\\LivreMaitriserEthereum\\adoc\\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html" \l "etherstore_attack) .

*Exemplo 2. Attack.sol*

import " EtherStore.sol ";

contrato Ataque {

EtherStore public etherStore ;

// inicializa a variável etherStore com o endereço do contrato

construtor( endereço \_ etherStoreAddress ) {

etherStore = EtherStore (\_ etherStoreAddress );

}

função attackEtherStore ( ) externo a pagar {

//ataca até o éter mais próximo

require( valor da msg >= 1 éter);

// envia eth para a função depositFunds ( )

etherStore.depositFunds. valor ( 1 éter)();

// inicia a mágica

etherStore.withdrawFunds (1 éter );

}

função coletarÉter ( ) public {

msg.sender .transfer ( this.balance ) ;

}

// função de fallback - onde a mágica acontece

função () a pagar {

if ( etherStore.balance > 1 ether) {

etherStore.withdrawFunds (1 éter );

}

}

}

Como a exploração pode ocorrer? Primeiro, o invasor criaria o contrato malicioso (digamos no endereço 0x0… ​123 ) com o endereço do contrato da EtherStore como o único parâmetro do construtor. Isso inicializaria e apontaria a variável pública etherStore para o contrato a ser atacado.

O invasor então chamaria a função attackEtherStore , com alguma quantidade de ether maior ou igual a 1 — vamos supor 1 ether por enquanto. Neste exemplo, também assumiremos que vários outros usuários depositaram ether neste contrato, de modo que seu saldo atual seja 10 ether . O seguinte vontade então ocorrer :

1. *Attack.sol* , linha 15: A função depositFunds do contrato EtherStore será chamada com um msg.value de 1 ether (e muito gás). O remetente ( msg.sender ) será o contrato malicioso ( 0x0… ​123 ). Assim , saldos[0x 0.. 123] = 1 éter .
2. *Attack.sol* , linha 17: O contrato malicioso chamará a função retirarFunds do contrato EtherStore com um parâmetro de 1 ether . Isso passará por todos os requisitos (linhas 12 a 16 do contrato EtherStore ), pois nenhuma retirada anterior foi feita.
3. *EtherStore.sol* , linha 17: O contrato enviará 1 ether de volta ao contrato malicioso.
4. *Attack.sol* , linha 25: O pagamento ao contrato malicioso executará a função de fallback.
5. *Attack.sol* , linha 26: O saldo total do contrato EtherStore era 10 ether e agora é 9 ether , então essa instrução if passa.
6. *Attack.sol* , linha 27: A função de fallback chama o EtherStore RetireFunds funciona novamente e ' redigita ' o contrato *EtherStore* .
7. *EtherStore.sol* , linha 11: Nesta segunda chamada para retirarFunds , o saldo do contrato atacante ainda é 1 ether pois a linha 18 ainda não foi executada. Assim, ainda temos saldos[0x 0.. 123] = 1 éter . Este também é o caso da variável lastWithdrawTime . Novamente , nós passar todos os requisitos .
8. *EtherStore.sol* , linha 17: O contrato atacante retira mais 1 ether .
9. etapas 4 a 8 são repetidas até que não seja mais o caso de EtherStore.balance > 1 , conforme ditado pela linha 26 em *Attack.sol* .
10. *Attack.sol* , linha 26: Quando houver 1 (ou menos) ether restante no contrato EtherStore , essa instrução if falhará. Isso permitirá que as linhas 18 e 19 do contrato EtherStore sejam executadas (para cada chamada à função retirarFunds ).
11. *EtherStore.sol* , linhas 18 e 19: Os saldos e mapeamentos lastWithdrawTime serão definidos e a execução será finalizada.

O resultado final é que o invasor retirou todos, exceto 1 ether do contrato EtherStore em uma única transação.

Técnicas Preventivas

Existem várias técnicas comuns que ajudam a evitar possíveis vulnerabilidades de reentrada em contratos inteligentes. A primeira é (sempre que possível) usar a função de [transferência](http://bit.ly/2Ogvnng) integrada ao enviar ether para contratos externos. A função de transferência só envia gás 2300 com a chamada externa, o que não é suficiente para que o endereço de destino/contrato chame outro contrato (ou seja, reinsira o contrato de envio).

A segunda técnica é garantir que toda a lógica que altera as variáveis de estado aconteça antes que o ether seja enviado para fora do contrato (ou qualquer chamada externa). No exemplo EtherStore , as linhas 18 e 19 de *EtherStore.sol* devem ser colocadas antes da linha 17. É uma boa prática que qualquer código que execute chamadas externas para endereços desconhecidos seja a última operação em uma função localizada ou execução de parte de código. Isso é conhecido como [padrão checks-effects-interactions](http://bit.ly/2EVo70v) .

Uma terceira técnica é introduzir um mutex — ou seja, adicionar uma variável de estado que bloqueia o contrato durante a execução do código, evitando chamadas reentrantes .

A aplicação de todas essas técnicas (usar as três é desnecessário, mas fazemos isso para fins demonstrativos) ao *EtherStore.sol* , fornece o contrato sem reentrância :

contrato EtherStore {

// inicializa o mutex

bool reEntrancyMutex = false;

uint256 retirada públicaLimit = 1 éter;

mapping( address => uint256) public lastWithdrawTime ;

mapeamento( endereço => uint256) saldos públicos;

função depositFunds ( ) externo a pagar {

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

}

função retireFunds (uint256 \_ weiToWithdraw ) public {

require (! reEntrancyMutex );

require(balances[ msg.sender ] >= \_ weiToWithdraw );

//limita a retirada

require( \_ weiToWithdraw <= retiradaLimit );

// limita o tempo permitido para retirar

require( now >= lastWithdrawTime [ msg.sender ] + 1 semanas);

saldos[ msg.sender ] -= \_ weiToWithdraw ;

lastWithdrawTime [ msg.sender ] = agora;

// define o mutex reEntrancy antes da chamada externa

reEntrancyMutex = true;

msg.sender .transfer ( \_ weiToWithdraw );

// libera o mutex após a chamada externa

reEntrancyMutex = false;

}

}

Exemplo do mundo real: O DAO

O ataque DAO (Organização Autônoma Descentralizada) foi um dos principais hacks que ocorreram no início do desenvolvimento do Ethereum. Na época, o contrato tinha mais de US$ 150 milhões. A reentrada desempenhou um papel importante no ataque, que acabou levando ao hard fork que criou o Ethereum Classic (ETC). Para uma boa análise do exploit DAO, consulte <http://bit.ly/2EQaLCI>. Mais informações sobre o histórico do fork do Ethereum, a linha do tempo do hack DAO e o nascimento do ETC em um hard fork podem ser encontradas em [[](file:///\\\\wsl.localhost\\Ubuntu-20.04\\home\\syr3fx\\Code\\github\\maitriser-ca\\LivreMaitriserEthereum\\adoc\\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html" \l "ethereum_standards) [ethereum\_standards ]](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#ethereum_standards) .

Sobrefluxos/subfluxos aritméticos

A Máquina Virtual Ethereum especifica tipos de dados de tamanho fixo para inteiros. Isso significa que uma variável inteira pode representar apenas um determinado intervalo de números. Um uint8 , por exemplo, só pode armazenar números no intervalo [0,255]. Tentar armazenar 256 em um uint8 resultará em 0 . Se não houver cuidado, as variáveis no Solidity podem ser exploradas se a entrada do usuário estiver desmarcada e forem realizados cálculos que resultem em números fora do intervalo do tipo de dados que os armazena.

Para ler mais sobre over/underflows aritméticos, consulte [“How to Secure Your Smart Contracts”](https://bit.ly/2nNLuOr) , [Ethereum Smart Contract Best Practices](https://bit.ly/2MOfBPv) e [“Ethereum, Solidity and integer overflows: Programming blockchains like 1970”](https://bit.ly/2xvbx1M) .

A vulnerabilidade

Um over/underflow ocorre quando uma operação é executada que requer uma variável de tamanho fixo para armazenar um número (ou parte dos dados) que está fora do intervalo do tipo de dados da variável.

Por exemplo, subtrair 1 de uma variável uint8 (inteiro sem sinal de 8 bits; ou seja, não negativo) cujo valor é 0 resultará no número 255 . Este é um *underflow* . Atribuímos um número abaixo do intervalo do uint8 , de modo que o resultado *envolve* e fornece o maior número que um uint8 pode armazenar. Da mesma forma, adicionar 2^8=256 a um uint8 deixará a variável inalterada, pois envolvemos todo o comprimento do uint . Duas analogias simples desse comportamento são hodômetros em carros, que medem a distância percorrida (eles voltam a 000000, após o maior número, ou seja, 999999, é superado) e funções matemáticas periódicas (adicionando 2 π ao argumento de sin deixa o valor inalterado ).

Adicionar números maiores que o intervalo do tipo de dados é chamado de *estouro* . Para maior clareza, adicionar 257 a um uint8 que atualmente tem um valor de 0 resultará no número 1 . Às vezes é instrutivo pensar em variáveis de tamanho fixo como sendo cíclicas, onde começamos novamente do zero se adicionarmos números acima do maior número armazenado possível e começamos a contagem regressiva do maior número se subtrairmos de zero. No caso de tipos int com sinal, que *podem* representar números negativos, recomeçamos quando atingimos o maior valor negativo; por exemplo, se tentarmos subtrair 1 de um int8 cujo valor é -128 , obteremos 127 .

Esses tipos de pegadinhas numéricas permitem que os invasores façam mau uso do código e criem fluxos lógicos inesperados. Por exemplo, considere o contrato [TimeLock em TimeLock.sol](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#timelock_sol_security) .

*Exemplo 3. TimeLock.sol*

contrato TimeLock {

mapping( address => uint ) saldos públicos;

mapping( address => uint ) public lockTime ;

função deposit( ) externo a pagar {

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

lockTime [ msg.sender ] = agora + 1 semanas;

}

função aumentarLockTime ( uint \_secondsToIncrease ) public {

lockTime [ msg.sender ] += \_ secondsToIncrease ;

}

função retirar( ) public {

require(balances[ msg.sender ] > 0);

require( now > lockTime [ msg.sender ]);

saldos[ msg.remetente ] = 0;

msg.sender .transfer ( saldo);

}

}

Este contrato foi projetado para funcionar como um cofre do tempo: os usuários podem depositar ether no contrato e ele ficará bloqueado por pelo menos uma semana. O usuário pode estender o tempo de espera para mais de 1 semana, se desejar, mas, uma vez depositado, o usuário pode ter certeza de que seu ether está bloqueado com segurança por pelo menos uma semana - ou assim pretende este contrato.

No caso de um usuário ser forçado a entregar sua chave privada, um contrato como esse pode ser útil para garantir que seu éter não possa ser obtido por um curto período de tempo. Mas se um usuário tivesse bloqueado 100 ether neste contrato e entregasse suas chaves a um invasor, o invasor poderia usar um estouro para receber o ether, independentemente do lockTime .

O invasor pode determinar o lockTime atual para o endereço para o qual ele agora possui a chave (é uma variável pública). Vamos chamar isso de userLockTime . Eles poderiam então chamar a função raiseLockTime e passar como argumento o número 2^256 - userLockTime . Esse número seria adicionado ao userLockTime atual e causaria um estouro, redefinindo lockTime [ msg.sender ] para 0 . O invasor pode simplesmente chamar a função de retirada para obter sua recompensa.

Vejamos outro exemplo ( [exemplo de vulnerabilidade Underflow do](file:///\\\\wsl.localhost\\Ubuntu-20.04\\home\\syr3fx\\Code\\github\\maitriser-ca\\LivreMaitriserEthereum\\adoc\\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html" \l "underflow_vulnerability_example_from_ethernaut_challenge) [desafio Ethernaut](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#underflow_vulnerability_example_from_ethernaut_challenge) ), este dos [desafios](https://github.com/OpenZeppelin/ethernaut) Ethernaut .

**ALERTA DE SPOILER:** *Se você ainda não fez os desafios do Ethernaut , isso dá uma solução para um dos níveis* .

*Exemplo 4. Exemplo de vulnerabilidade de underflow do desafio Ethernaut*

solidez do pragma ^0,4. 18;

token de contrato {

mapping( address => uint ) balances;

uint public totalSupply ;

function Token( uint \_ initialSupply ) {

balances[ msg.sender ] = totalSupply = \_ initialSupply ;

}

function transfer( address \_to, uint \_value) public return (bool) {

require(balances[ msg.sender ] - \_value >= 0);

balances[ msg.sender ] -= \_value;

saldos[\_to] += \_ valor;

retorne verdadeiro;

}

function balanceOf ( address \_owner) public constante retorna ( uint balance) {

retornar saldos[\_proprietário ];

}

}

Este é um contrato de token simples que emprega uma função de transferência , permitindo que os participantes movam seus tokens. Você pode ver o erro neste contrato?

A falha vem na função de transferência . A instrução require na linha 13 pode ser ignorada usando um underflow. Considere um usuário com saldo zero. Eles podem chamar a função de transferência com qualquer \_value diferente de zero e passar a instrução require na linha 13. Isso ocorre porque balances[ msg.sender ] é 0 (e um uint256 ), portanto, subtrair qualquer valor positivo (excluindo 2^256 ) resultará em um número positivo, como descrito anteriormente. Isso também vale para a linha 14, onde o saldo será creditado com um número positivo. Assim, neste exemplo, um invasor pode obter tokens gratuitos devido a uma vulnerabilidade de underflow.

Técnicas Preventivas

A técnica convencional atual para se proteger contra vulnerabilidades de over/overflow é usar ou construir bibliotecas matemáticas que substituem os operadores matemáticos padrão adição, subtração e multiplicação (a divisão é excluída porque não causa over/underflows e o EVM reverte na divisão por 0 ).

[O OpenZeppelin](https://github.com/OpenZeppelin/openzeppelin-solidity) fez um ótimo trabalho construindo e auditando bibliotecas seguras para a comunidade Ethereum. Em especial, sua A [biblioteca](http://bit.ly/2ABhb4l) SafeMath pode ser usada para evitar vulnerabilidades de under/overflow.

Para demonstrar como essas bibliotecas são usadas no Solidity, vamos corrigir o contrato TimeLock usando a biblioteca SafeMath . A versão sem estouro do contrato é:

biblioteca SafeMath {

function mul ( uint256 a, uint256 b) retornos puros internos (uint256) {

if (a == 0) {

retornar 0;

}

uint256 c = a \* b;

assert( c/a == b);

retornar c;

}

function div( uint256 a, uint256 b) retornos puros internos (uint256) {

// assert( b > 0); // Solidity é lançado automaticamente ao dividir por 0

uint256 c = a / b;

// assert( a == b \* c + a % b); // Isso vale em todos os casos

retornar c;

}

function sub( uint256 a, uint256 b) retornos puros internos (uint256) {

assert( b <= a);

retornar a - b;

}

function add( uint256 a, uint256 b) retornos puros internos (uint256) {

uint256 c = a + b;

assert( c >= a);

retornar c;

}

}

contrato TimeLock {

usando SafeMath para uint ; // usa a biblioteca para o tipo uint

mapeamento( endereço => uint256) saldos públicos;

mapping( address => uint256) public lockTime ;

função deposit( ) externo a pagar {

balances[ msg.sender ] = balances[ msg.sender ].add( msg.value );

lockTime [ msg.sender ] = now.add (1 semanas);

}

função aumentarLockTime ( uint256 \_secondsToIncrease ) public {

lockTime [ msg.sender ] = lockTime [ msg.sender ].add(\_ secondsToIncrease );

}

função retirar( ) public {

require(balances[ msg.sender ] > 0);

require( now > lockTime [ msg.sender ]);

saldos[ msg.remetente ] = 0;

msg.sender .transfer ( saldo);

}

}

Observe que todas as operações matemáticas padrão foram substituídas por aquelas definidas na biblioteca SafeMath . O contrato TimeLock não executa mais nenhuma operação que seja capaz de under/overflow.

Exemplos do mundo real: PoWHC e estouro de transferência em lote (CVE-2018–10299)

Proof of Weak Hands Coin ( PoWHC ), originalmente concebido como uma espécie de piada, era um esquema Ponzi escrito por um coletivo da internet. Infelizmente , parece que o(s) autor(es) do contrato não haviam visto over/underflows antes e, consequentemente, 866 ether foram liberados de seu contrato. Eric Banisadr dá uma boa visão geral de como ocorreu o underflow (que não é muito diferente do desafio Ethernaut descrito anteriormente) em seu [blog](https://bit.ly/2wrxIFJ) sobre o evento.

[Outro exemplo](http://bit.ly/2CUf7WG) vem da implementação de uma função batchTransfer ( ) em um grupo de contratos de token ERC20. A implementação continha uma vulnerabilidade de estouro; você pode ler sobre os detalhes na [conta](https://bit.ly/2HDlIs8) do PeckShield .

Éter inesperado

Normalmente, quando o ether é enviado para um contrato, ele deve executar a função de fallback ou outra função definida no contrato. Existem duas exceções a isso, onde o ether pode existir em um contrato sem que nenhum código seja executado. Os contratos que dependem da execução de código para todo o ether enviado a eles podem ser vulneráveis a ataques em que o ether é enviado à força.

Para ler mais sobre isso, consulte [“Como proteger seus contratos inteligentes”](https://bit.ly/2MR8Gp0) e [“Padrões de segurança de solidez - forçando o Ether a um contrato”](http://bit.ly/2RjXmUWl) .

A vulnerabilidade

Uma técnica de programação defensiva comum que é útil para impor transições de estado corretas ou validar operações é *a verificação invariável* . Essa técnica envolve definir um conjunto de invariantes (métricas ou parâmetros que não devem ser alterados) e verificar se eles permanecem inalterados após uma única (ou muitas) operação(ões). Este é tipicamente um bom design, desde que as invariantes que estão sendo verificadas sejam de fato invariantes. Um exemplo de invariante é a oferta total de uma emissão fixa [Token ERC20](http://bit.ly/2CUf7WG) . Como nenhuma função deve modificar essa invariante, pode-se adicionar uma verificação à função de transferência que garanta que totalSupply permaneça inalterado, para garantir que a função esteja funcionando conforme o esperado.

Em particular, há um invariante aparente que pode ser tentador usar, mas que pode de fato ser manipulado por usuários externos (independentemente das regras estabelecidas no contrato inteligente). Este é o éter atual armazenado no contrato. Muitas vezes, quando os desenvolvedores aprendem Solidity , eles têm a concepção errônea de que um contrato só pode aceitar ou obter ether por meio de funções pagáveis. Esse equívoco pode levar a contratos com falsas suposições sobre o equilíbrio de éter dentro deles, o que pode levar a uma série de vulnerabilidades. A arma fumegante para esta vulnerabilidade é o uso (incorreto) de this.balance .

Existem duas maneiras pelas quais o ether pode (forçadamente) ser enviado para um contrato sem usar uma função de pagamento ou executar qualquer código no contrato:

**Autodestruição/suicídio**

Qualquer contrato é capaz de implementar a [função](http://bit.ly/2RovrDf) selfdestruct , que remove todo o bytecode do endereço do contrato e envia todo o éter armazenado lá para o endereço especificado pelo parâmetro. Se esse endereço especificado também for um contrato, nenhuma função (incluindo o fallback) será chamada. Portanto, a função de autodestruição pode ser usada para enviar éter à força para qualquer contrato, independentemente de qualquer código que possa existir no contrato, mesmo contratos sem funções pagáveis. Isso significa que qualquer invasor pode criar um contrato com uma função de autodestruição , enviar ether para ela, chamar selfdestruct (destino) e forçar o envio de ether para um contrato de destino . Martin Swende tem uma excelente [postagem no blog](http://bit.ly/2OfLukM) descrevendo algumas peculiaridades do opcode de autodestruição (Quirk #2) junto com uma conta de como os nós clientes estavam verificando invariantes incorretas, o que poderia ter levado a uma falha bastante catastrófica da rede Ethereum.

**Éter pré-enviado**

Outra maneira de inserir o ether em um contrato é pré-carregar o endereço do contrato com o ether. Os endereços de contrato são determinísticos — na verdade, o endereço é calculado a partir do hash Keccak-256 (comumente sinônimo de SHA-3) do endereço que cria o contrato e do nonce da transação que cria o contrato. Especificamente, é do formato address = sha3( rlp.encode ([ account\_address,transaction\_nonce ])) (veja a discussão de Adrian Manning sobre [“Keyless Ether”](http://bit.ly/2EPj5Tq) para alguns casos de uso divertidos disso). Isso significa que qualquer pessoa pode calcular qual será o endereço de um contrato antes de ser criado e enviar ether para esse endereço. Quando o contrato for criado, ele terá um saldo de éter diferente de zero.

Vamos explorar algumas armadilhas que podem surgir com esse conhecimento. Considere o contrato excessivamente simples em [EtherGame.sol](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#etherGame_security) .

*Exemplo 5. EtherGame.sol*

contrato EtherGame {

uint public payoutMileStone1 = 3 ether;

uint public mileStone1Reward = 2 ether;

uint public payoutMileStone2 = 5 éter;

uint public mileStone2Reward = 3 ether;

uint public finalMileStone = 10 éter;

uint public finalRecompensa = 5 ether;

mapping( address => uint ) redeemableEther ;

// Os usuários pagam 0,5 ether. Em marcos específicos, credite suas contas.

função play( ) externa a pagar {

require( valor da msg == 0,5 éter); // cada jogada é 0,5 éter

uint currentBalance = this.balance + msg.value ;

// garante que não haja jogadores após o jogo terminar

require( currentBalance <= finalMileStone );

// se estiver em um marco, credita a conta do jogador

if ( currentBalance == payoutMileStone1) {

resgatávelEther [ msg.sender ] += mileStone1Reward;

}

else if ( currentBalance == payoutMileStone2) {

resgatáveisEther [ msg.sender ] += mileStone2Reward;

}

else if ( currentBalance == finalMileStone ) {

redemableEther [ msg.sender ] += finalReward ;

}

Retorna;

}

função reclamaçãoRecompensa ( ) public {

// assegura que o jogo está completo

require( this.balance == finalMileStone );

// garante que há uma recompensa para dar

require( resgatávelEther [ msg.sender ] > 0);

éter resgatável [ msg.remetente ] = 0;

msg.sender .transfer ( transferValue ) ;

}

}

Este contrato representa um jogo simples (que naturalmente envolveria condições de corrida) onde os jogadores enviam 0,5 éter para o contrato na esperança de ser o jogador que atinge um dos três marcos primeiro. Os marcos são denominados em éter. O primeiro a atingir o marco pode reivindicar uma parte do éter quando o jogo terminar. O jogo termina quando o marco final (10 éteres) é alcançado; os usuários podem então reivindicar suas recompensas.

Os problemas com o contrato EtherGame vêm do mau uso deste equilíbrio nas duas linhas 14 (e por associação 16) e 32. Um invasor malicioso pode enviar à força uma pequena quantidade de éter - digamos, 0,1 éter - por meio da função de autodestruição ( discutido anteriormente) para evitar que futuros jogadores atinjam um marco. this.balance nunca será um múltiplo de 0,5 ether graças a esta contribuição de 0,1 ether, porque todos os jogadores legítimos só podem enviar incrementos de 0,5 ether. Isso impede que todas as condições if nas linhas 18, 21 e 24 sejam verdadeiras.

Pior ainda, um atacante vingativo que perdesse um marco poderia enviar à força 10 ether (ou uma quantidade equivalente de ether que eleva o saldo do contrato acima do finalMileStone ), o que bloquearia todas as recompensas no contrato para sempre. Isso ocorre porque a função ClaimReward sempre será revertida, devido ao require na linha 32 (ou seja, porque this.balance é maior que finalMileStone ).

Técnicas Preventivas

Esse tipo de vulnerabilidade geralmente surge do mau uso deste.balance . A lógica do contrato, quando possível, deve evitar depender de valores exatos do saldo do contrato, pois pode ser manipulada artificialmente. Se aplicar a lógica com base em this.balance , você terá que lidar com saldos inesperados.

Se forem necessários valores exatos de éter depositado, deve ser usada uma variável autodefinida que é incrementada em funções a pagar, para rastrear com segurança o éter depositado. Esta variável não será influenciada pelo éter forçado enviado por meio de uma chamada de autodestruição .

Com isso em mente, uma versão corrigida do contrato EtherGame poderia ser assim:

contrato EtherGame {

uint public payoutMileStone1 = 3 ether;

uint public mileStone1Reward = 2 ether;

uint public payoutMileStone2 = 5 éter;

uint public mileStone2Reward = 3 ether;

uint public finalMileStone = 10 éter;

uint public finalRecompensa = 5 ether;

uint public depositedWei ;

mapeamento (endereço => uint ) redeemableEther ;

função play( ) externa a pagar {

require( valor da msg == 0,5 éter);

uint currentBalance = depositedWei + msg. valor ;

// garante que não haja jogadores após o jogo terminar

require( currentBalance <= finalMileStone );

if ( currentBalance == payoutMileStone1) {

resgatávelEther [ msg.sender ] += mileStone1Reward;

}

else if ( currentBalance == payoutMileStone2) {

resgatáveisEther [ msg.sender ] += mileStone2Reward;

}

else if ( currentBalance == finalMileStone ) {

redemableEther [ msg.sender ] += finalReward ;

}

depositedWei += msg. valor ;

Retorna;

}

função reclamaçãoRecompensa ( ) public {

// assegura que o jogo está completo

require( depositedWei == finalMileStone );

// garante que há uma recompensa para dar

require( resgatávelEther [ msg.sender ] > 0);

éter resgatável [ msg.remetente ] = 0;

msg.sender .transfer ( transferValue ) ;

}

}

Aqui, criamos uma nova variável, depositedWei , que rastreia o éter conhecido depositado, e é essa variável que usamos para nossos testes. Observe que não temos mais nenhuma referência a this.balance .

Outros exemplos

Alguns exemplos de contratos exploráveis foram dados no [Underhanded Solidity Coding Contest](https://github.com/Arachnid/uscc/tree/master/submissions-2017/) , que também fornece exemplos estendidos de várias armadilhas levantadas nesta seção.

CHAMADA DELEGADA

Os opcodes CALL e DELEGATECALL são úteis para permitir que os desenvolvedores do Ethereum modularizem seu código. Chamadas de mensagens externas padrão para contratos são tratadas pelo opcode CALL , por meio do qual o código é executado no contexto do contrato/função externa. O opcode DELEGATECALL é quase idêntico, exceto que o código executado no endereço de destino é executado no contexto do contrato de chamada e msg.sender e msg.value permanecem inalterados. Esse recurso permite a implementação de *bibliotecas* , permitindo que os desenvolvedores implantem código reutilizável uma vez e o chamem de contratos futuros.

Embora as diferenças entre esses dois opcodes sejam simples e intuitivas, o uso de DELEGATECALL pode levar à execução inesperada do código.

Para ler mais, veja Loi.Luu's [Pergunta do Ethereum Stack Exchange sobre este tópico](http://bit.ly/2AAElb8) e os [documentos do Solidity](http://bit.ly/2Oi7UlH) .

A vulnerabilidade

Como resultado da natureza de preservação de contexto de DELEGATECALL , construir bibliotecas personalizadas livres de vulnerabilidades não é tão fácil quanto se imagina. O código nas próprias bibliotecas pode ser seguro e livre de vulnerabilidades; no entanto, quando executado no contexto de outro aplicativo, novas vulnerabilidades podem surgir. Vamos ver um exemplo bastante complexo disso, usando números de Fibonacci.

Considere a biblioteca em [FibonacciLib.sol](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#fibonacci_security) , que pode gerar a sequência de Fibonacci e sequências de forma semelhante. (Nota: este código foi modificado de <https://bit.ly/2MReuii>.)

*Exemplo 6. FibonacciLib.sol*

// contrato de biblioteca - calcula números semelhantes a Fibonacci

contrato FibonacciLib {

// inicializando a sequência padrão de Fibonacci

uint início público ;

uint publiccalculadoFibNumber ; \_

// modifica o número zero na sequência

função setStart ( uint \_start) public {

inicio = \_ inicio;

}

function setFibonacci ( uint n) public {

calculadoFibNumber = fibonacci (n );

}

function fibonacci ( uint n) retornos internos ( uint ) {

if (n == 0) return início;

senão se (n == 1) return start + 1;

else return fibonacci ( n - 1) + fibonacci (n - 2);

}

}

Esta biblioteca fornece uma função que pode gerar o *n* - º número de Fibonacci na sequência. Ele permite que os usuários alterem o número inicial da sequência ( start ) e calculem os *n* - º números semelhantes a Fibonacci nesta nova sequência.

Vamos agora considerar um contrato que utiliza esta biblioteca, mostrado em [FibonacciBalance.sol](file:///\\\\wsl.localhost\\Ubuntu-20.04\\home\\syr3fx\\Code\\github\\maitriser-ca\\LivreMaitriserEthereum\\adoc\\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html" \l "fib_balance_security) .

*Exemplo 7. FibonacciBalance.sol*

contrato FibonacciBalance {

endereço público fibonacciLibrary ;

// o número Fibonacci atual a ser retirado

uint publiccalculadoFibNumber ; \_

// o número de sequência de Fibonacci inicial

uint public start = 3;

uint retirada públicaCounter ;

// o seletor de função Fibonancci

bytes4 constante fibSig = bytes4(sha3(" setFibonacci (uint256)") );

// construtor - carrega o contrato com ether

construtor( endereço \_ fibonacciLibrary ) externo a pagar {

fibonacciLibrary = \_ fibonacciLibrary ;

}

função retirar( ) {

contador de retirada += 1;

// calcula o número de Fibonacci para o usuário de saque atual-

// isso define ocalcFibNumber

require( fibonacciLibrary.delegatecall ( fibSig , retiradaCounter ));

msg.sender .transfer ( calculadoFibNumber \* 1 éter);

}

// permite que os usuários chamem as funções da biblioteca Fibonacci

função( ) public {

require( fibonacciLibrary.delegatecall ( msg.data ));

}

}

Este contrato permite que um participante retire ether do contrato, sendo a quantidade de ether igual ao número de Fibonacci correspondente à ordem de retirada do participante; ou seja, o primeiro participante recebe 1 éter, o segundo também recebe 1, o terceiro recebe 2, o quarto recebe 3, o quinto recebe 5 e assim por diante (até que o saldo do contrato seja menor que o número de Fibonacci que está sendo retirado).

Há uma série de elementos neste contrato que podem exigir alguma explicação. Em primeiro lugar, há uma variável de aparência interessante, fibSig . Isso contém os primeiros 4 bytes do hash Keccak-256 (SHA-3) da string ' setFibonacci (uint256)' . Isso é conhecido como [seletor de função](http://bit.ly/2RmueMP) e é colocado em calldata para especificar qual função de um contrato inteligente será chamada. Ele é usado na função delegatecall na linha 21 para especificar que desejamos executar a função fibonacci (uint256) . O segundo argumento em delegatecall é o parâmetro que estamos passando para a função. Em segundo lugar, assumimos que o endereço para a biblioteca FibonacciLib está corretamente referenciado no construtor ( [External Contract Referencing](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#external_contract_referencing) discute algumas vulnerabilidades potenciais relacionadas a esse tipo de inicialização de referência de contrato).

Você consegue identificar algum erro neste contrato? Se alguém implantar este contrato, preenchê-lo com éter e chamar retirar , ele provavelmente será revertido.

Você deve ter notado que a variável de estado start é usada tanto na biblioteca quanto no contrato de chamada principal. No contrato da biblioteca, start é usado para especificar o início da sequência de Fibonacci e é definido como 0 , enquanto é definido como 3 no contrato de chamada. Você também deve ter notado que a função de fallback no contrato FibonacciBalance permite que todas as chamadas sejam passadas para o contrato de biblioteca, o que permite que a função setStart do contrato de biblioteca seja chamada. Lembrando que preservamos o estado do contrato, pode parecer que esta função permitiria alterar o estado da variável start no contrato local FibonnacciBalance . Nesse caso, isso permitiria retirar mais éter, pois o númeroFibNúmero calculado resultante depende da variável inicial (conforme visto no contrato da biblioteca). Na verdade , a função setStart não modifica (e não pode) modificar a variável start no contrato FibonacciBalance . A vulnerabilidade subjacente neste contrato é significativamente pior do que apenas modificar a variável inicial .

Antes de discutir o problema real, vamos fazer um rápido desvio para entender como as variáveis de estado são realmente armazenadas nos contratos. As variáveis de estado ou armazenamento (variáveis que persistem em transações individuais) são colocadas em *slots* sequencialmente à medida que são introduzidas no contrato. (Existem algumas complexidades aqui; consulte os [documentos do Solidity](http://bit.ly/2JslDWf) para um entendimento mais completo.)

Como exemplo, vejamos o contrato da biblioteca. Ele tem duas variáveis de estado, start e calculadaFibNumber . A primeira variável, start , é armazenada no armazenamento do contrato no slot[ 0] (ou seja, o primeiro slot). A segunda variável, calculadaFibNumber , é colocada no próximo slot de armazenamento disponível, slot[ 1] . A função setStart recebe uma entrada e define start para qualquer que seja a entrada. Esta função, portanto, define slot[ 0] para qualquer entrada que fornecermos na função setStart . Da mesma forma, a função setFibonacci define calculadoFibNumber para o resultado de fibonacci (n) . Novamente, isso é simplesmente definir o slot de armazenamento [ 1] para o valor de fibonacci (n) .

Agora vamos olhar para o contrato FibonacciBalance . Storage slot[ 0] agora corresponde ao endereço fibonacciLibrary e slot[1] corresponde a CalculateFibNumber . É nesse mapeamento incorreto que ocorre a vulnerabilidade. chamada de delegado *preserva o contexto do contrato* . Isso significa que o código executado via delegatecall atuará no estado (ou seja, armazenamento) do contrato de chamada.

Agora observe que em retirar na linha 21 executamos fibonacciLibrary.delegatecall ( fibSig,withdrawalCounter ) . Isso chama a função setFibonacci , que, como discutimos, modifica o slot de armazenamento[ 1] , que em nosso contexto atual é calculadoFibNumber . Isso é o esperado (ou seja, após a execução, o CalculateFibNumber é modificado). No entanto, lembre-se de que a variável inicial no contrato FibonacciLib está localizada no slot de armazenamento [ 0] , que é o endereço fibonacciLibrary no contrato atual. Isso significa que a função fibonacci dará um resultado inesperado. Isso ocorre porque ele faz referência a start ( slot[ 0] ), que no contexto de chamada atual é o endereço fibonacciLibrary (que geralmente será bastante grande, quando interpretado como um uint ). Assim , é provável que a função de retirada seja revertida, pois não conterá a quantidade uint ( fibonacciLibrary ) de éter, que é o que CalculateFibNumber retornará.

Pior ainda, o contrato FibonacciBalance permite que os usuários chamem todas as funções fibonacciLibrary por meio da função fallback na linha 26. Como discutimos anteriormente, isso inclui a função setStart . Discutimos que esta função permite que qualquer pessoa modifique ou defina o slot de armazenamento [ 0] . Nesse caso, o slot de armazenamento [ 0] é o endereço fibonacciLibrary . Portanto, um invasor pode criar um contrato malicioso, converter o endereço em um uint (isso pode ser feito em Python facilmente usando int('<address>',16) ) e chamar setStart (< attack\_contract\_address\_as\_uint >) . Isso mudará fibonacciLibrary para o endereço do contrato de ataque. Então, sempre que um usuário chama retirar ou a função de fallback, o contrato malicioso será executado (o que pode roubar todo o saldo do contrato) porque modificamos o endereço real de fibonacciLibrary . Um exemplo de tal contrato de ataque seria:

contrato Ataque {

uint storageSlot0; // corresponde a fibonacciLibrary

uint storageSlot1; // corresponde aocalcFibNumber

// fallback - isso será executado se uma função especificada não for encontrada

função( ) public {

armazenamentoSlot1 = 0; // definimos calculadoFibNumber como 0, portanto, se retirar

// é chamado não enviamos nenhum éter

< attacker\_address > .transfer ( this.balance ); // pegamos todo o éter

}

}

Observe que este contrato de ataque modifica o CalculateFibNumber alterando o slot de armazenamento[ 1] . Em princípio, um invasor pode modificar quaisquer outros slots de armazenamento que escolher, para realizar todos os tipos de ataques a esse contrato. Incentivamos você a colocar esses contratos no [Remix](https://remix.ethereum.org) e experimentar diferentes contratos de ataque e mudanças de estado por meio dessas funções de chamada de delegação .

Também é importante notar que quando dizemos que delegatecall é de preservação de estado, não estamos falando sobre os nomes das variáveis do contrato, mas sim dos slots de armazenamento reais para os quais esses nomes apontam. Como você pode ver neste exemplo, um simples erro pode levar um invasor a seqüestrar todo o contrato e seu éter.

Técnicas Preventivas

O Solidity fornece a palavra-chave library para implementar contratos de biblioteca (consulte os [documentos](http://bit.ly/2zjD8TI) para obter mais detalhes). Isso garante que o contrato da biblioteca seja sem estado e não autodestrutível . Forçar bibliotecas a serem sem estado reduz as complexidades do contexto de armazenamento demonstradas nesta seção. As bibliotecas sem estado também evitam ataques em que os invasores modificam o estado da biblioteca diretamente para afetar os contratos que dependem do código da biblioteca. Como regra geral , ao usar DELEGATECALL , preste muita atenção ao possível contexto de chamada do contrato de biblioteca e do contrato de chamada e, sempre que possível, construa bibliotecas sem estado.

Exemplo do mundo real: carteira multisig de paridade (segundo hack)

O hack Second Parity Multisig Wallet é um exemplo de como o código de biblioteca bem escrito pode ser explorado se executado fora do contexto pretendido. Existem várias boas explicações sobre esse hack, como [“Parity Multisig Hacked. Novamente”](http://bit.ly/2Dg7GtW) e [“Um olhar aprofundado sobre o bug multisig de paridade ”](http://bit.ly/2Of06B9) .

Para adicionar a essas referências, vamos explorar os contratos que foram explorados. Os contratos de biblioteca e carteira podem ser encontrados [no GitHub](http://bit.ly/2OgnXQC) .

O contrato da biblioteca é o seguinte:

contrato WalletLibrary é WalletEvents {

...

// jogue a menos que o contrato ainda não tenha sido inicializado.

modificador only\_uninitialized { if ( m\_numOwners > 0) throw; \_; }

// construtor - apenas passe o array proprietário para multipropriedade e

// o limite para daylimit

function initWallet ( address[ ] \_owners, uint \_required, uint \_ daylimit )

only\_uninitialized {

initDaylimit ( \_daylimit );

initMultiowned ( \_owners, \_required);

}

// mata o contrato enviando tudo para `\_to`.

function kill( address \_to) onlymanyowners (sha3( msg.data )) external {

suicídio(\_to );

}

...

}

E aqui está o contrato da carteira:

contrato Carteira é WalletEvents {

...

// MÉTODOS

// é chamado quando nenhuma outra função corresponde

função ( ) a pagar {

// acabou de receber algum dinheiro?

if ( valor da msg > 0)

Deposit( msg.remetente , msg.valor );

senão se ( msg. data.length > 0)

\_ walletLibrary.delegatecall ( msg.data );

}

...

// CAMPOS

endereço constante \_ walletLibrary =

0xcafecafecafecafecafecafecafecafecafe;

}

Observe que o contrato Wallet basicamente passa todas as chamadas para o contrato WalletLibrary por meio de uma chamada delegada. O endereço constante \_ walletLibrary neste trecho de código atua como um espaço reservado para o realmente implantado Contrato WalletLibrary (que estava em 0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4 ).

A operação pretendida desses contratos era ter um contrato Wallet implantável simples e de baixo custo , cuja base de código e funcionalidade principal estivessem no contrato WalletLibrary . Infelizmente, o contrato WalletLibrary é em si um contrato e mantém seu próprio estado. Você pode ver por que isso pode ser um problema?

É possível enviar chamadas para o próprio contrato WalletLibrary . Especificamente, o contrato WalletLibrary pode ser inicializado e se tornar propriedade. Na verdade, um usuário fez isso, chamando a função initWallet no contrato WalletLibrary e tornando-se proprietário do contrato da biblioteca. O mesmo usuário posteriormente chamou a função kill . Como o usuário era um proprietário do contrato da biblioteca, o modificador foi aprovado e o contrato da biblioteca se autodestruiu. Como todos os contratos da Wallet existentes se referem a este contrato de biblioteca e não contêm nenhum método para alterar essa referência, todas as suas funcionalidades, incluindo a capacidade de retirar o ether, foram perdidas junto com o contrato da WalletLibrary . Como resultado, todo o éter em todas as carteiras multisig Parity desse tipo se tornou instantaneamente perdido ou permanentemente irrecuperável.

Visibilidades padrão

As funções no Solidity têm especificadores de visibilidade que determinam como elas podem ser chamadas. A visibilidade determina se uma função pode ser chamada externamente por usuários, por outros contratos derivados, apenas internamente ou apenas externamente. Existem quatro especificadores de visibilidade, que são descritos em detalhes nos [documentos do Solidity](http://bit.ly/2ABiv7j) . As funções são padronizadas como public , permitindo que os usuários as chamem externamente. Veremos agora como o uso incorreto de especificadores de visibilidade pode levar a algumas vulnerabilidades devastadoras em contratos inteligentes.

A vulnerabilidade

A visibilidade padrão para funções é public , portanto, funções que não especificam sua visibilidade serão chamadas por usuários externos. O problema surge quando os desenvolvedores omitem erroneamente os especificadores de visibilidade em funções que deveriam ser privadas (ou apenas chamáveis dentro do próprio contrato).

Vamos explorar rapidamente um exemplo trivial:

contrato HashForEther {

função retirarGanhos ( ) {

// Vencedor se os últimos 8 caracteres hexadecimais do endereço forem 0

require(uint32( msg.sender ) == 0);

\_ enviarGanhos ( );

}

função \_ enviarGanhos ( ) {

msg.sender .transfer ( this.balance ) ;

}

}

Este contrato simples foi projetado para atuar como um jogo de recompensas de adivinhação de endereços. Para ganhar o saldo do contrato, o usuário deve gerar um endereço Ethereum cujos últimos 8 caracteres hexadecimais sejam 0. Uma vez alcançado, ele pode chamar a função retirarWinnings para obter sua recompensa.

Infelizmente, a visibilidade das funções não foi especificada. Em particular, a função \_ sendWinnings é pública (o padrão) e, portanto, qualquer endereço pode chamar essa função para roubar a recompensa.

Técnicas Preventivas

É uma boa prática sempre especificar a visibilidade de todas as funções em um contrato, mesmo que sejam intencionalmente públicas . Versões recentes do solc mostram um aviso para funções que não possuem visibilidade explícita definida, para incentivar essa prática.

Exemplo do mundo real: carteira multisig de paridade (primeiro hack)

No primeiro hack multisig da Parity , cerca de US$ 31 milhões em Ether foram roubados, principalmente de três carteiras. Uma boa recapitulação de exatamente como isso foi feito é dada por [Haseeb Qureshi](https://bit.ly/2vHiuJQ) .

Essencialmente, a carteira multisig é construída a partir de um contrato Wallet básico , que chama um contrato de biblioteca contendo a funcionalidade principal (conforme descrito em [Exemplo do mundo real: Parity](file:///\\\\wsl.localhost\\Ubuntu-20.04\\home\\syr3fx\\Code\\github\\maitriser-ca\\LivreMaitriserEthereum\\adoc\\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html" \l "multisig_secondhack) [Multisig Wallet (Second Hack)](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#multisig_secondhack) ). O contrato da biblioteca contém o código para inicializar a carteira, como pode ser visto no seguinte trecho:

contrato WalletLibrary é WalletEvents {

...

// MÉTODOS

...

// construtor recebe o número de sigs necessários para fazer protected

// transações " onlymanyowners " , bem como a seleção de endereços

// capaz de confirmá-los

function initMultiowned ( address[ ] \_owners, uint \_required) {

m\_numOwners = \_ proprietários.comprimento + 1;

m\_ proprietários [ 1] = uint ( msg.sender );

m\_ownerIndex [ uint ( msg.sender )] = 1;

para ( uint i = 0; i < \_ proprietários.comprimento ; ++ e )

{

m\_ proprietários [ 2 + i ] = uint (\_proprietários[ i ]);

m\_ownerIndex [ uint (\_owners[ i ])] = 2 + i ;

}

m\_requerido = \_ obrigatório;

}

...

// construtor - apenas passe o array proprietário para multipropriedade e

// o limite para daylimit

function initWallet ( address[ ] \_owners, uint \_required, uint \_ daylimit ) {

initDaylimit ( \_daylimit );

initMultiowned ( \_owners, \_required);

}

}

Observe que nenhuma das funções especifica sua visibilidade, portanto, ambas são padronizadas como public . A função initWallet é chamada no construtor da carteira e define os proprietários da carteira multisig , como pode ser visto na função initMultiowned . Como essas funções foram acidentalmente deixadas públicas , um invasor conseguiu chamar essas funções em contratos implantados, redefinindo a propriedade para o endereço do invasor. Sendo o proprietário, o invasor esvaziou as carteiras de todo o seu éter.

Ilusão de Entropia

Todas as transações na blockchain Ethereum são operações de transição de estado determinísticas. Isso significa que cada transação modifica o estado global do ecossistema Ethereum de forma calculável, sem incertezas. Isso tem a implicação fundamental de que não há fonte de entropia ou aleatoriedade no Ethereum. Alcançar a entropia descentralizada (aleatoriedade) é um problema bem conhecido para o qual muitas soluções foram propostas, incluindo [RANDAO](https://github.com/randao/randao) , ou usando uma cadeia de hashes, conforme descrito por Vitalik Buterin na postagem do blog [“Validator Ordering and Randomness in PoS ”](https://vitalik.ca/files/randomness.html) .

A vulnerabilidade

Alguns dos primeiros contratos construídos na plataforma Ethereum foram baseados em jogos de azar. Fundamentalmente, o jogo requer incerteza (algo em que apostar), o que torna a construção de um sistema de jogo no blockchain (um sistema determinístico) bastante difícil. É claro que a incerteza deve vir de uma fonte externa ao blockchain. Isso é possível para apostas entre jogadores (veja por exemplo a [técnica commit–reveal](http://bit.ly/2CUh2KS) ); no entanto, é significativamente mais difícil se você deseja implementar um contrato para atuar como “a casa” (como no blackjack ou na roleta). Uma armadilha comum é usar variáveis de bloco futuras, ou seja, variáveis contendo informações sobre o bloco de transação cujos valores ainda não são conhecidos, como hashes, carimbos de data/hora, números de bloco ou limites de gás. O problema com isso é que eles são controlados pelo minerador que minera o bloco e, como tal, não são verdadeiramente aleatórios. Considere, por exemplo, um contrato inteligente de roleta com lógica que retorna um número preto se o próximo hash de bloco terminar em um número par. Um minerador (ou pool de mineradores) pode apostar US$ 1 milhão no preto. Se eles resolverem o próximo bloco e encontrarem o hash terminar em um número ímpar, eles poderiam felizmente não publicar seu bloco e minerar outro, até encontrarem uma solução com o hash do bloco sendo um número par (supondo que a recompensa e as taxas do bloco sejam menores que US$ 1 milhão). Usar variáveis passadas ou presentes pode ser ainda mais devastador, como Martin Swende demonstra em seu excelente [post no blog](http://martin.swende.se/blog/Breaking_the_house.html) . Além disso, usar apenas variáveis de bloco significa que o número pseudo-aleatório será o mesmo para todas as transações em um bloco, para que um invasor possa multiplicar seus ganhos fazendo muitas transações dentro de um bloco (se houver uma aposta máxima).

Técnicas Preventivas

A fonte de entropia (aleatoriedade) deve ser externa ao blockchain. Isso pode ser feito entre pares com sistemas como [commit–reveal](http://bit.ly/2CUh2KS) ou através da mudança do modelo de confiança para um grupo de participantes (como em [RandDAO](https://github.com/randao/randao) ). Isso também pode ser feito por meio de uma entidade centralizada que atua como um oráculo de aleatoriedade. Variáveis de bloco (em geral, existem algumas exceções) não devem ser usadas para fonte de entropia, pois podem ser manipuladas por mineradores.

Exemplo do mundo real: contratos PRNG

Em fevereiro de 2018 Arseny Reutov [blogou](http://bit.ly/2Q589lx) sobre sua análise de 3.649 contratos inteligentes ao vivo que estavam usando algum tipo de gerador de números pseudoaleatórios (PRNG); ele encontrou 43 contratos que poderiam ser explorados.

Referenciamento de contrato externo

Um dos benefícios do “computador mundial” Ethereum é a capacidade de reutilizar código e interagir com contratos já implantados na rede. Como resultado, um grande número de contratos faz referência a contratos externos, geralmente por meio de chamadas de mensagens externas. Essas chamadas de mensagens externas podem mascarar as intenções de agentes mal-intencionados de algumas maneiras não óbvias, que examinaremos agora.

A vulnerabilidade

No Solidity, qualquer endereço pode ser convertido em um contrato, independentemente de o código no endereço representar o tipo de contrato que está sendo convertido. Isso pode causar problemas, especialmente quando o autor do contrato está tentando ocultar um código malicioso. Vamos ilustrar isso com um exemplo.

Considere um pedaço de código como [Rot13Encryption.sol](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#rot13_security) , que implementa rudimentarmente a [cifra ROT13](https://en.wikipedia.org/wiki/ROT13) .

*Exemplo 8. Rot13Encryption.sol*

// contrato de criptografia

contrato Rot13Encryption {

evento Resultado( string convertString );

// rot13-criptografa uma string

function rot13Encrypt (texto de string) public {

uint256 comprimento = bytes(texto ).comprimento ;

for (var i = 0; i < comprimento; i ++) {

byte char = bytes(texto)[ i ];

// assembly embutido para modificar a string

conjunto {

//pega o primeiro byte

char : = byte(0,char)

// se o caractere estiver em [ n,z ], ou seja, envolvendo

if e( gt (char,0x6D), lt (char,0x7B))

// subtrai do número ASCII 'a',

// a diferença entre o caractere <char> e 'z'

{ char := sub(0x60, sub(0x7A,char)) }

if iszero ( eq(char, 0x20)) // ignora espaços

// adiciona 13 ao caractere

{mstore8(add(add(text,0x20), mul (i,1)), add(char,13))}

}

}

emita Resultado(texto );

}

// rot13-descriptografa uma string

function rot13Decrypt (texto de string) public {

uint256 comprimento = bytes(texto ).comprimento ;

for (var i = 0; i < comprimento; i ++) {

byte char = bytes(texto)[ i ];

conjunto {

char : = byte(0,char)

if e( gt (char,0x60), lt (char,0x6E))

{ char := add(0x7B, sub(char,0x61)) }

if iszero ( eq(char, 0x20))

{mstore8(add(add(text,0x20), mul (i,1)), sub(char,13))}

}

}

emita Resultado(texto );

}

}

Esse código simplesmente pega uma string (letras a–z, sem validação) e a *criptografa* deslocando cada caractere 13 lugares para a direita (envolvendo z ); ou seja, um desloca para n e x desloca para k . O assembly no contrato anterior não precisa ser entendido para apreciar a questão que está sendo discutida, então os leitores não familiarizados com o assembly podem ignorá-lo com segurança.

Agora considere o seguinte contrato, que usa esse código para sua criptografia:

import "Rot13Encryption.sol ";

// criptografa suas informações ultra-secretas

contrato CriptografiaContrato {

// biblioteca para criptografia

Rot13 Criptografia de criptografiaLibrary ;

// construtor - inicializa a biblioteca

constructor( Rot13Encryption \_ encryptionLibrary ) {

biblioteca de criptografia = \_ biblioteca de criptografia ;

}

function encryptPrivateData ( string privateInfo ) {

// potencialmente fazer algumas operações aqui

criptografiaLibrary.rot13Encrypt( privateInfo );

}

}

O problema com este contrato é que o endereço da biblioteca de criptografia não é público ou constante. Assim, o implantador do contrato poderia dar um endereço no construtor que aponta para este contrato:

// contrato de criptografia

contrato Rot26Encryption {

evento Resultado( string convertString );

// rot13-criptografa uma string

function rot13Encrypt (texto de string) public {

uint256 comprimento = bytes(texto ).comprimento ;

for (var i = 0; i < comprimento; i ++) {

byte char = bytes(texto)[ i ];

// assembly embutido para modificar a string

conjunto {

//pega o primeiro byte

char : = byte(0,char)

// se o caractere estiver em [ n,z ], ou seja, envolvendo

if e( gt (char,0x6D), lt (char,0x7B))

// subtrai do número ASCII 'a',

// a diferença entre o caractere <char> e 'z'

{ char := sub(0x60, sub(0x7A,char)) }

// ignora espaços

if iszero ( eq(char, 0x20))

// adiciona 26 ao caractere!

{mstore8(add(add(text,0x20), mul (i,1)), add(char,26))}

}

}

emita Resultado(texto );

}

// rot13-descriptografa uma string

function rot13Decrypt (texto de string) public {

uint256 comprimento = bytes(texto ).comprimento ;

for (var i = 0; i < comprimento; i ++) {

byte char = bytes(texto)[ i ];

conjunto {

char : = byte(0,char)

if e( gt (char,0x60), lt (char,0x6E))

{ char := add(0x7B, sub(char,0x61)) }

if iszero ( eq(char, 0x20))

{mstore8(add(add(text,0x20), mul (i,1)), sub(char,26))}

}

}

emita Resultado(texto );

}

}

Este contrato implementa a cifra ROT26, que muda cada caractere em 26 lugares (ou seja, não faz nada). Novamente, não há necessidade de entender a montagem neste contrato. Mais simplesmente, o invasor poderia ter vinculado o seguinte contrato ao mesmo efeito:

contrato {

evento Print( texto string);

function rot13 Criptografar ( string text) public {

emita Imprimir(texto );

}

}

Se o endereço de qualquer um desses contratos fosse fornecido no construtor, a função encryptPrivateData simplesmente produziria um evento que imprime os dados privados não criptografados.

Embora neste exemplo um contrato semelhante a uma biblioteca tenha sido definido no construtor, geralmente é o caso de um usuário privilegiado (como um proprietário) poder alterar os endereços de contrato da biblioteca. Se um contrato vinculado não contiver a função que está sendo chamada, a função de fallback será executada. Por exemplo, com a linha encryptionLibrary.rot13 ​Encrypt ( ) , se o contrato especificado por encryptionLibrary for:

contrato em branco {

evento Print( texto string);

função () {

emit Print("Aqui" );

// coloque código malicioso aqui e ele será executado

}

}

então um evento com o texto Aqui seria emitido. Assim, se os usuários podem alterar bibliotecas de contrato, eles podem, em princípio, fazer com que outros usuários executem código arbitrário sem saber.

|  |  |
| --- | --- |
| **AVISO** | Os contratos aqui representados são apenas para fins demonstrativos e não representam criptografia adequada. Elas não deve ser usado para criptografia . |

Técnicas Preventivas

Conforme demonstrado anteriormente, contratos seguros podem (em alguns casos) ser implantados de forma a se comportarem de forma maliciosa. Um auditor pode verificar publicamente um contrato e fazer com que seu proprietário o implante de maneira maliciosa, resultando em um contrato auditado publicamente com vulnerabilidades ou intenção maliciosa.

Existem várias técnicas que impedem esses cenários.

Uma técnica é usar a palavra-chave new para criar contratos. No exemplo anterior, o construtor pode ser escrito como:

construtor( ) {

criptografiaLibrary = new Rot13 Encryption( );

}

Dessa forma, uma instância do contrato referenciado é criada no momento da implantação e o implantador não pode substituir o contrato Rot13Encryption sem alterá-lo.

Outra solução é codificar endereços de contratos externos.

Em geral, o código que chama contratos externos deve sempre ser auditado com cuidado. Como desenvolvedor, ao definir contratos externos, pode ser uma boa ideia tornar públicos os endereços do contrato (o que não é o caso do exemplo do pote de mel na seção a seguir) para permitir que os usuários examinem facilmente o código referenciado pelo contrato. Por outro lado, se um contrato tiver um endereço de contrato de variável privada, pode ser um sinal de alguém se comportando maliciosamente (como mostrado no exemplo do mundo real). Se um usuário puder alterar um endereço de contrato usado para chamar funções externas, pode ser importante (em um contexto de sistema descentralizado) implementar um mecanismo de bloqueio de tempo e/ou votação para permitir que os usuários vejam qual código está sendo alterado ou para dar aos participantes a chance de optar por entrar/não com o novo endereço do contrato.

Exemplo do mundo real: pote de mel de reentrada

Vários potes de mel recentes foram lançados na rede principal . Esses contratos tentam enganar os hackers do Ethereum que tentam explorar os contratos, mas que, por sua vez, acabam perdendo o ether para o contrato que esperam explorar. Um exemplo emprega esse ataque substituindo um contrato esperado por um mal-intencionado no construtor. O código pode ser encontrado [aqui](http://bit.ly/2JtdqRi) :

solidez do pragma ^0,4. 19;

contrato Private\_Bank

{

mapeamento (endereço => uint ) saldos públicos;

uint public MinDeposit = 1 éter;

Log TransferLog ;

função Private\_ Bank ( endereço \_log)

{

TransferLog = Log(\_log );

}

função Depósito ( )

público

a pagar

{

if( valor da msg >= MinDeposit )

{

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

TransferLog.AddMessage ( msg.sender ,msg.value,"Depósito ");

}

}

função CashOut ( uint \_am)

{

if(\_am<=balances[ msg.sender ])

{

if( msg.sender .call.value (\_am)())

{

saldos[ msg.remetente ]-=\_am;

TransferLog.AddMessage ( msg.sender ,\_am," CashOut ");

}

}

}

function( ) pagamento externo{}

}

registro de contrato

{

mensagem de estrutura

{

endereço do remetente;

seqüência de dados ;

uint Val;

uint Tempo ;

}

Message[ ] história pública;

Mensagem LastMsg ;

função AddMessage ( endereço \_ adr, uint \_ val, string \_data)

público

{

LastMsg.Sender = \_ adr ;

LastMsg.Time = agora;

LastMsg.Val = \_val ;

LastMsg.Data = \_ data;

History.push ( LastMsg );

}

}

Esta [postagem](http://bit.ly/2Q58VyX) de um usuário do reddit explica como eles perderam 1 ether para este contrato tentando explorar o bug de reentrada que esperavam estar presente no contrato.

Endereço curto/ataque de parâmetro

Esse ataque não é realizado nos próprios contratos do Solidity, mas em aplicativos de terceiros que possam interagir com eles. Esta seção foi adicionada para completar e dar ao leitor uma noção de como os parâmetros podem ser manipulados em contratos.

Para ler mais, consulte [“The ERC20 Short Address Attack Explained”](http://bit.ly/2yKme14) , [“ICO Smart Contract Vulnerability: Short Address Attack”](http://bit.ly/2yFOGRQ) ou esta [postagem do Reddit](http://bit.ly/2CQjBhc) .

A vulnerabilidade

Ao passar parâmetros para um contrato inteligente, os parâmetros são codificados de acordo com a [especificação ABI](http://bit.ly/2Q5VIG9) . É possível enviar parâmetros codificados menores que o comprimento esperado do parâmetro (por exemplo, enviar um endereço que tenha apenas 38 caracteres hexadecimais (19 bytes) em vez dos 40 caracteres hexadecimais padrão (20 bytes)). Nesse cenário, o EVM adicionará zeros ao final dos parâmetros codificados para compor o comprimento esperado.

Isso se torna um problema quando aplicativos de terceiros não validam entradas. O exemplo mais claro é uma exchange que não verifica o endereço de um token ERC20 quando um usuário solicita uma retirada. Este exemplo é abordado com mais detalhes no post de Peter Vessenes , [“O ataque de endereço curto ERC20 explicado”](http://bit.ly/2Q1ybpQ) .

Considere a interface de função de transferência [ERC20 padrão](http://bit.ly/2CUf7WG) , observando a ordem dos parâmetros:

função transfer( endereço para, tokens uint ) public retorna (bool sucesso);

Agora considere uma troca que detém uma grande quantidade de um token (digamos REP ) e um usuário que deseja retirar sua parte de 100 tokens. O usuário enviaria seu endereço, 0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead , e o número de tokens, 100 . A troca codificaria esses parâmetros na ordem especificada pela função de transferência ; ou seja, endereço então tokens . O resultado codificado seria:

a9059cbb000000000000000000000000deaddeaddea \

dmortomortomortomortomortomorto0000000000000

00000000000000000000000000000000056bc75e2d63100000

Os primeiros 4 bytes ( a9059cbb ) são a transferência [function signature/selector](http://bit.ly/2RmueMP) , os próximos 32 bytes são o endereço e os 32 bytes finais representam o número uint256 de tokens. Observe que o hexadecimal 56bc75e2d63100000 no final corresponde a 100 tokens (com 18 casas decimais, conforme especificado pelo contrato de token REP ).

Vejamos agora o que aconteceria se alguém enviasse um endereço que estivesse faltando 1 byte (2 dígitos hexadecimais). Especificamente, digamos que um invasor envie 0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde como um endereço (faltando os dois últimos dígitos) e os mesmos 100 tokens para retirar. Se a exchange não validar essa entrada, ela será codificada como:

a9059cbb000000000000000000000000deaddeaddea \

dmortomortomortomortomortomorto00000000000000

0000000000000000000000000000000056bc75e2d6310000000

A diferença é sutil. Observe que 00 foi adicionado ao final da codificação, para compensar o endereço curto que foi enviado. Quando isso for enviado para o contrato inteligente, os parâmetros de endereço serão lidos como 0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde00 e o valor será lido como 56bc75e2d6310000000 (observe os dois 0s extras). Esse valor agora é de 25.600 tokens (o valor foi multiplicado por 256 ). Neste exemplo, se a exchange tivesse tantos tokens, o usuário retiraria 25.600 tokens (enquanto a exchange pensa que o usuário está retirando apenas 100 ) para o endereço modificado. Obviamente , o invasor não possuirá o endereço modificado neste exemplo, mas se o invasor gerar qualquer endereço que termine em 0s (o que pode ser facilmente forçado) e usar esse endereço gerado, ele poderá roubar tokens da troca desavisada .

Técnicas Preventivas

Todos os parâmetros de entrada em aplicativos externos devem ser validados antes de enviá-los para o blockchain. Deve-se notar também que a ordenação dos parâmetros desempenha um papel importante aqui. Como o preenchimento ocorre apenas no final, a ordenação cuidadosa dos parâmetros no contrato inteligente pode mitigar algumas formas desse ataque.

Valores de retorno CALL não verificados

Existem várias maneiras de realizar chamadas externas no Solidity. O envio de ether para contas externas geralmente é realizado pelo método de transferência . No entanto, a função send também pode ser usada e, para chamadas externas mais versáteis, o opcode CALL pode ser empregado diretamente no Solidity. As funções call e send retornam um booleano indicando se a chamada foi bem-sucedida ou falhou. Assim, essas funções têm uma ressalva simples, em que a transação que executa essas funções não será revertida se a chamada externa ( inicializada por call ou send ) falhar; em vez disso, as funções simplesmente retornarão false . Um erro comum é que o desenvolvedor espera que ocorra uma reversão se a chamada externa falhar e não verifica o valor de retorno.

Para ler mais, veja o nº 4 no [DASP Top 10 de 2018](http://www.dasp.co/#item-4) e [“Scanning Live Ethereum Contracts for the 'Unchecked-Send' Bug”](http://bit.ly/2RnS1vA) .

A vulnerabilidade

Considere o seguinte exemplo:

contrato Loto {

bool public payedOut = false;

endereço público vencedor;

uint public winAmount ;

// ... funcionalidade extra aqui

função sendToWinner ( ) public {

exigir (! pagoOut );

vencedor.send ( winAmount );

desembolsado = true;

}

função retirarLeftOver ( ) public {

exigir( pagoSaída );

msg.sender .send ( this.balance );

}

}

Isso representa um contrato do tipo Lotto, onde um vencedor recebe winAmount of Ether, o que normalmente deixa um pouco de sobra para qualquer um sacar.

A vulnerabilidade existe na linha 11, onde um envio é usado sem verificar a resposta. Neste exemplo trivial, um vencedor cuja transação falha (por ficar sem gás ou por ser um contrato que lança intencionalmente a função de fallback) permite que payedOut seja definido como verdadeiro , independentemente de o ether ter sido enviado ou não. Neste caso, qualquer pessoa pode retirar os ganhos do vencedor através da função retirarLeftOver .

Técnicas Preventivas

Sempre que possível, use a função de transferência em vez de enviar , pois a transferência será revertida se a transação externa for revertida. Caso seja necessário enviar , verifique sempre o valor de retorno.

[recomendação](http://bit.ly/2CSdF7y) mais robusta é adotar um *padrão de retirada* . Nesta solução, cada usuário deve chamar uma função de retirada isolada que trata do envio de ether fora do contrato e lida com as consequências de transações de envio com falha. A ideia é isolar logicamente a funcionalidade de envio externo do restante da base de código e colocar o ônus de uma transação potencialmente falha no usuário final que está chamando a função de retirada.

Exemplo do mundo real: Etherpot e Rei do Éter

[Etherpot](http://bit.ly/2OfHalK) era uma loteria de contrato inteligente, não muito diferente do contrato de exemplo mencionado anteriormente. A queda deste contrato foi principalmente devido ao uso incorreto de hashes de bloco (apenas os últimos 256 hashes de bloco são utilizáveis; veja o [post de](http://bit.ly/2Jpzf4x) Aakil Fernandes sobre como o Etherpot falhou em levar isso em conta corretamente). No entanto, este contrato também sofreu com um valor de chamada não verificado. Considere a função cash in [lotto.sol : Code snippet](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#lotto_security) .

*Exemplo 9. lotto.sol : trecho de código*

...

função dinheiro( uint roundIndex , uint subpotIndex ){

var subpotsCount = getSubpotsCount ( roundIndex );

if( subpotIndex >= subpotsCount )

Retorna;

var DecisionBlockNumber = getDecisionBlockNumber ( roundIndex,subpotIndex );

if( decisãoNúmeroBloco > número.bloco )

Retorna;

if(rounds[ roundIndex ]. isCashed [ subpotIndex ])

Retorna;

// Os subpots só podem ser sacados uma vez. Isso é para evitar pagamentos duplos

var vencedor = calcularVencedor ( roundIndex,subpotIndex );

var subpot = getSubpot ( roundIndex );

vencedor.send ( subpote );

rodadas[ roundIndex ]. isCashed [ subpotIndex ] = true;

//Marca a rodada como descontada

}

...

Observe que na linha 21 o valor de retorno da função de envio não é verificado, e a linha a seguir define um booleano indicando que o vencedor recebeu seus fundos. Este bug pode permitir um estado onde o ganhador não receba seu ether, mas o estado do contrato pode indicar que o ganhador já foi pago.

Uma versão mais séria desse bug ocorreu no [King of the Ether](http://bit.ly/2ACsfi1) . Um excelente [post-mortem](http://bit.ly/2ESoaub) deste contrato foi escrito detalhando como um envio com falha não verificado pode ser usado para atacar o contrato.

Condições de Corrida/Corrida de Frente

A combinação de chamadas externas para outros contratos e a natureza multiusuário do blockchain subjacente dá origem a uma variedade de potenciais armadilhas do Solidity, pelas quais os usuários *correm* a execução do código para obter estados inesperados. A reentrância (discutida anteriormente neste capítulo) é um exemplo de tal condição de corrida. Nesta seção, discutiremos outros tipos de condições de corrida que podem ocorrer na blockchain Ethereum. Há uma variedade de boas postagens sobre esse assunto, incluindo “Condições de corrida” no [Ethereum Wiki](http://bit.ly/2yFesFF) , [#7 no DASP Top10 de 2018](http://www.dasp.co/#item-7) e as [melhores práticas de contrato inteligente da Ethereum](http://bit.ly/2Q6E4lP) .

A vulnerabilidade

Como na maioria das blockchains, os nós Ethereum agrupam transações e as formam em blocos. As transações só são consideradas válidas quando um minerador resolve um mecanismo de consenso (atualmente [Ethash](http://bit.ly/2yI5Dv7) PoW para Ethereum). O minerador que resolve o bloco também escolhe quais transações do pool serão incluídas no bloco, normalmente ordenadas pelo gasPrice de cada transação. Aqui está um possível vetor de ataque. Um invasor pode observar o pool de transações em busca de transações que possam conter soluções para problemas e modificar ou revogar as permissões do solucionador ou alterar o estado em um contrato prejudicial ao solucionador. O invasor pode então obter os dados dessa transação e criar uma transação própria com um gasPrice mais alto para que sua transação seja incluída em um bloco antes do original.

Vamos ver como isso pode funcionar com um exemplo simples. Considere o contrato mostrado em [FindThisHash.sol](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#findthishash_security) .

*Exemplo 10. FindThisHash.sol*

contrato FindThisHash {

bytes32 hash público constante =

0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

constructor( ) externo a pagar {} // carrega com ether

function solve( solução de string) public {

// Se você encontrar a pré-imagem do hash, receba 1000 ether

require( hash == sha3(solução));

msg.sender .transfer ( 1000 éter);

}

}

Digamos que este contrato contenha 1.000 éter . O usuário que pode encontrar a pré-imagem do seguinte hash SHA-3:

0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a

pode enviar a solução e recuperar os 1.000 éteres . Digamos que um usuário descubra que a solução é o Ethereum! . Eles chamam resolver com Ethereum! como parâmetro. Infelizmente, um invasor foi esperto o suficiente para observar o pool de transações para qualquer pessoa que enviasse uma solução. Eles veem essa solução, verificam sua validade e enviam uma transação equivalente com um gasPrice muito maior do que a transação original. O minerador que resolver o bloco provavelmente dará preferência ao invasor devido ao gasPrice mais alto e minerar sua transação antes da do solucionador original. O invasor pegará os 1.000 ether e o usuário que resolveu o problema não receberá nada. Lembre-se de que, nesse tipo de vulnerabilidade "front-running", os mineradores são incentivados exclusivamente a executar os ataques (ou podem ser subornados para executar esses ataques com taxas extravagantes). A possibilidade de o invasor ser um minerador não deve ser subestimada.

Técnicas Preventivas

Existem duas classes de atores que podem executar esses tipos de ataques front-running: usuários (que modificam o gasPrice de suas transações) e os próprios mineradores (que podem reordenar as transações em um bloco como acharem melhor). Um contrato que é vulnerável à primeira classe (usuários) é significativamente pior do que um vulnerável à segunda (mineradores), pois os mineradores só podem realizar o ataque quando resolvem um bloco, o que é improvável para qualquer minerador individual visando um bloco específico . Aqui listaremos algumas medidas de mitigação relativas a ambas as classes de invasores.

Um método é colocar um limite superior no gasPrice . Isso evita que os usuários aumentem o gasPrice e obtenham pedidos de transações preferenciais além do limite superior. Essa medida protege apenas contra a primeira classe de invasores (usuários arbitrários). Os mineradores nesse cenário ainda podem atacar o contrato, pois podem ordenar as transações em seu bloco como quiserem, independentemente do preço do gás.

Um método mais robusto é usar um esquema [commit-reveal .](http://bit.ly/2CUh2KS) Esse esquema determina que os usuários enviem transações com informações ocultas (normalmente um hash). Após a transação ter sido incluída em um bloco, o usuário envia uma transação revelando os dados que foram enviados (fase de revelação). Esse método impede que mineradores e usuários executem transações de frente, pois eles não podem determinar o conteúdo da transação. Este método, no entanto, não pode ocultar o valor da transação (que em alguns casos é a informação valiosa que precisa ser ocultada). O contrato inteligente do [ENS](https://ens.domains/) permitiu que os usuários enviassem transações cujos dados confirmados incluíam a quantidade de ether que eles estavam dispostos a gastar. Os usuários poderiam então enviar transações de valor arbitrário. Durante a fase de revelação, os usuários eram reembolsados da diferença entre o valor enviado na transação e o valor que estavam dispostos a gastar.

Outra sugestão de Lorenz Breidenbach, Phil Daian , Ari Juels e Florian Tramèr é usar [“submarine sends”](http://bit.ly/2SygqQx) . Uma implementação eficiente dessa ideia requer o opcode CREATE2 , que atualmente não foi adotado, mas parece provável que esteja nos próximos hard forks.

Exemplos do mundo real: ERC20 e Bancor

O [padrão ERC20](http://bit.ly/2CUf7WG) é bastante conhecido por construir tokens no Ethereum. Esse padrão tem uma potencial vulnerabilidade de execução frontal que ocorre devido à função de aprovação . [Mikhail Vladimirov e Dmitry Khovratovich](http://bit.ly/2DbvQpJ) escreveram uma boa explicação sobre essa vulnerabilidade (e formas de mitigar o ataque).

A norma especifica a função de aprovação como:

função aprovar( endereço \_spender, uint256 \_value) retorna (bool sucesso)

Esta função permite que um usuário permita que outros usuários transfiram tokens em seu nome. A vulnerabilidade de execução frontal ocorre no cenário em que uma usuária Alice *aprova* que seu amigo Bob gaste 100 tokens. Mais tarde, Alice decide que quer revogar a aprovação de Bob para gastar, digamos, 100 tokens, então ela cria uma transação que define a alocação de Bob para 50 tokens. Bob, que tem observado cuidadosamente a cadeia, vê essa transação e constrói uma transação própria gastando os 100 tokens. Ele coloca um gasPrice em sua transação mais alto do que o de Alice, então prioriza sua transação sobre a dela. Algumas implementações de aprovar permitiriam que Bob transferisse seus 100 tokens e então, quando a transação de Alice fosse confirmada, redefinisse a aprovação de Bob para 50 tokens, dando a Bob acesso a 150 tokens.

Outro exemplo proeminente do mundo real é o [Bancor](https://www.bancor.network/) . Ivan Bogatyy e sua equipe documentaram um ataque lucrativo à implementação inicial do Bancor . Sua [postagem no blog](http://bit.ly/2EUlLzb) e [a palestra do DevCon3](http://bit.ly/2yHgkhs) discutem em detalhes como isso foi feito. Essencialmente, os preços dos tokens são determinados com base no valor da transação; os usuários podem observar o pool de transações para transações do Bancor e administrá-las para lucrar com as diferenças de preço. Este ataque foi abordado pela equipe do Bancor .

Negação de serviço (DoS)

Essa categoria é muito ampla, mas consiste fundamentalmente em ataques em que os usuários podem tornar um contrato inoperante por um período de tempo ou, em alguns casos, permanentemente. Isso pode prender o ether nesses contratos para sempre, como foi o caso no [exemplo do mundo real: Parity Multisig Wallet (Second Hack)](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#multisig_secondhack) .

A vulnerabilidade

Existem várias maneiras pelas quais um contrato pode se tornar inoperável. Aqui destacamos apenas alguns padrões de codificação Solidity menos óbvios que podem levar a vulnerabilidades DoS:

**Fazendo um loop por meio de mapeamentos ou matrizes manipulados externamente**

Esse padrão geralmente aparece quando um proprietário deseja distribuir tokens para investidores com uma função de distribuição , como neste contrato de exemplo:

contrato DistribuirTokens {

endereço proprietário público; // é definido em algum lugar

endereço[ ] investidores; // array de investidores

uint [ ] investidorTokens ; // a quantidade de tokens que cada investidor recebe

// ... funcionalidade extra, incluindo transfertoken ( )

função investir( ) externa a pagar {

investidores.push ( msg.sender );

investidorTokens.push ( msg.value \* 5); // 5 vezes o wei enviado

}

função distribuir( ) public {

require( msg.sender == dono); // único dono

para ( uint i = 0; i < investidores.comprimento ; eu ++) {

// aqui transferToken ( to,amount ) transfere "amount" de

// tokens para o endereço "to"

transferToken (investidores[ i ], investidorTokens [ i ]);

}

}

}

Observe que o loop neste contrato é executado em um array que pode ser inflado artificialmente. Um invasor pode criar muitas contas de usuário, aumentando a matriz de investidores . Em princípio, isso pode ser feito de modo que o gás necessário para executar o loop for exceda o limite de gás do bloco, essencialmente tornando a função de distribuição inoperável.

**Operações do proprietário**

Outro padrão comum é quando os proprietários têm privilégios específicos nos contratos e devem executar alguma tarefa para que o contrato prossiga para o próximo estado. Um exemplo seria um contrato de Oferta Inicial de Moedas (ICO) que exige que o proprietário finalize o contrato, o que permite que os tokens sejam transferíveis. Por exemplo:

bool public isFinalized = false;

endereço proprietário público; // é definido em algum lugar

função finalize( ) public {

require( msg.sender == dono);

isFinalizado == true;

}

// ... funcionalidade extra de ICO

//função de transferência sobrecarregada

function transfer( address \_to, uint \_value) retorna (bool) {

require( isFinalizado );

super.transfer (\_ to,\_value )

}

...

Nesses casos, se o usuário privilegiado perder suas chaves privadas ou ficar inativo, todo o contrato de token ficará inoperante. Nesse caso, se o proprietário não puder chamar finalize , nenhum token poderá ser transferido; toda a operação do ecossistema de tokens depende de um único endereço.

**Estado em andamento com base em chamadas externas**

Às vezes, os contratos são escritos de tal forma que o progresso para um novo estado requer o envio de ether para um endereço ou a espera de alguma entrada de uma fonte externa. Esses padrões podem levar a ataques DoS quando a chamada externa falha ou é impedida por motivos externos. No exemplo de envio de ether, um usuário pode criar um contrato que não aceita ether. Se um contrato exige que o éter seja retirado para progredir para um novo estado (considere um contrato de bloqueio de tempo que exige que todo o éter seja retirado antes de poder ser usado novamente), o contrato nunca atingirá o novo estado, pois o éter nunca pode ser enviado para o contrato do usuário que não aceita ether.

Técnicas Preventivas

No primeiro exemplo, os contratos não devem percorrer estruturas de dados que podem ser manipuladas artificialmente por usuários externos. Um padrão de retirada é recomendado, em que cada um dos investidores chama uma função de retirada para reivindicar tokens de forma independente.

No segundo exemplo, um usuário privilegiado foi obrigado a alterar o estado do contrato. Em tais exemplos, um failsafe pode ser usado no caso de o proprietário ficar incapacitado. Uma solução é tornar o proprietário um contrato multisig . Outra solução é usar um bloqueio de tempo: no exemplo dado o require na linha 5 poderia incluir um mecanismo baseado em tempo, como require( msg.sender == owner || now > unlockTime ) , que permite que qualquer usuário finalize após um período de tempo especificado por unlockTime . Esse tipo de técnica de mitigação também pode ser usado no terceiro exemplo. Se chamadas externas forem necessárias para progredir para um novo estado, considere sua possível falha e, potencialmente, adicione uma progressão de estado baseada em tempo caso a chamada desejada nunca chegue.

|  |  |
| --- | --- |
| **NOTA** | Claro, existem alternativas centralizadas para essas sugestões: pode-se adicionar um MaintenanceUser que pode vir e corrigir problemas com vetores de ataque baseados em DoS, se necessário. Normalmente , esses tipos de contratos têm problemas de confiança, devido ao poder de tal entidade. |

Exemplos do mundo real: Governamental

[Governamental](http://governmental.github.io/GovernMental/) era um antigo esquema Ponzi que acumulava uma quantidade bastante grande de éter (1.100 éter, em um ponto). Infelizmente, ele era suscetível às vulnerabilidades DoS mencionadas nesta seção. Um [post do Reddit](http://bit.ly/2DcgvFc) por etherik descreve como o contrato exigia a exclusão de um grande mapeamento para retirar o ether. A exclusão desse mapeamento teve um custo de gás que ultrapassou o limite de gás de bloco na época e, portanto, não foi possível retirar o éter 1.100 . The contract address is [0xF45717552f12Ef7cb65e95476F217Ea008167Ae3](http://bit.ly/2Oh8j7R) , and you can see from transaction [0x0d80d67202bd9cb6773df8dd2020e719&thinsp;0a1b0793e8ec4fc105257e8128f0506b](http://bit.ly/2Ogzrnn) that the 1,100 ether were finally obtained with a transaction that used 2.5M gas (when the block gas limit had risen enough to allow such a transaction).

Manipulação de carimbo de data/hora do bloco

Os carimbos de data e hora de bloco têm sido usados historicamente para uma variedade de aplicações, como entropia para números aleatórios (consulte a [Ilusão de Entropia](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#entropyillusion_security) para obter mais detalhes), bloqueio de fundos por períodos de tempo e várias instruções condicionais de mudança de estado que dependem do tempo. Os mineradores têm a capacidade de ajustar ligeiramente os carimbos de data/hora, o que pode ser perigoso se os carimbos de data/hora do bloco forem usados incorretamente em contratos inteligentes.

Referências úteis para isso incluem [os documentos Solidity](http://bit.ly/2OdUC9C) e [a pergunta Ethereum Stack Exchange de Joris Bontje](http://bit.ly/2CQ8gh4) sobre o tópico.

A vulnerabilidade

block.timestamp e seu alias agora podem ser manipulados por mineradores se eles tiverem algum incentivo para fazê-lo. Vamos construir um jogo simples, mostrado em [roulette.sol](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#roulette_security) , que seria vulnerável à exploração de mineradores.

*Exemplo 11. roleta.sol*

roleta de contrato {

uint public pastBlockTime ; // força uma aposta por bloco

construtor( ) pagamento externo {} // contrato de financiamento inicial

// função de fallback usada para fazer uma aposta

função () externo a pagar {

require( valor da msg == 10 éter); // deve enviar 10 ether para jogar

require( now != pastBlockTime ); // apenas 1 transação por bloco

pastBlockTime = agora;

if( now % 15 == 0) { // vencedor

msg.sender .transfer ( this.balance ) ;

}

}

}

Este contrato se comporta como uma simples loteria. Uma transação por bloco pode apostar 10 ether por uma chance de ganhar o saldo do contrato. A suposição aqui é que os dois últimos dígitos do ` block.timestamp são distribuídos uniformemente. Se fosse esse o caso, haveria uma chance de 1 em 15 de ganhar esta loteria.

No entanto, como sabemos, os mineradores podem ajustar o timestamp caso precisem. Nesse caso em particular, se houver pools de ether suficientes no contrato, um minerador que resolve um bloco é incentivado a escolher um timestamp tal que block.timestamp ou agora módulo 15 seja 0 . Ao fazer isso, eles podem ganhar o éter bloqueado neste contrato junto com a recompensa do bloco. Como há apenas uma pessoa autorizada a apostar por bloco, isso também é vulnerável a ataques front-running (veja [Condições de Corrida/Front Running](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#frontrunning_security) para mais detalhes).

Na prática, os timestamps de bloco estão aumentando monotonicamente e, portanto, os mineradores não podem escolher timestamps de bloco arbitrários (eles devem ser posteriores a seus predecessores). Eles também estão limitados a definir tempos de bloqueio não muito distantes no futuro, pois esses blocos provavelmente serão rejeitados pela rede (nós não validarão blocos cujos carimbos de data e hora estão no futuro).

Técnicas Preventivas

Os timestamps de bloco não devem ser usados para entropia ou geração de números aleatórios – ou seja, eles não devem ser o fator decisivo (seja diretamente ou através de alguma derivação) para ganhar um jogo ou mudar um estado importante.

A lógica sensível ao tempo às vezes é necessária; por exemplo, para desbloquear contratos (time-locking), completar uma ICO após algumas semanas ou impor datas de expiração. Às vezes, é recomendável usar [block.number](http://bit.ly/2OdUC9C) e um tempo médio de bloco para estimar os tempos; com um tempo de bloqueio de 10 segundos , 1 semana equivale a aproximadamente 60480 blocos . Assim, especificar um número de bloco para alterar o estado do contrato pode ser mais seguro, pois os mineradores não conseguem manipular facilmente o número do bloco. O contrato [BAT ICO](http://bit.ly/2AAebFr) empregou essa estratégia.

Isso pode ser desnecessário se os contratos não estiverem particularmente preocupados com as manipulações do minerador do registro de data e hora do bloco, mas é algo a ser observado ao desenvolver contratos.

Exemplo do mundo real: Governamental

[Governamental](http://governmental.github.io/GovernMental/) , o antigo esquema Ponzi mencionado acima, também era vulnerável a um ataque baseado em timestamp. O contrato pago ao jogador que foi o último jogador a entrar (por pelo menos um minuto) em uma rodada. Assim, um minerador que era um jogador poderia ajustar o timestamp (para um tempo futuro, para fazer parecer que um minuto havia se passado) para fazer parecer que ele foi o último jogador a entrar por mais de um minuto (mesmo que isso não tenha sido verdade na realidade). Mais detalhes sobre isso podem ser encontrados no [post “History of Ethereum Security Vulnerabilities, Hacks and Their Fixes”](http://bit.ly/2Q1AMA6) de Tanya Bahrynovska .

Construtores com cuidado

Construtores são funções especiais que geralmente executam tarefas críticas e privilegiadas ao inicializar contratos. Antes do Solidity v0.4.22, construtores eram definidos como funções que tinham o mesmo nome do contrato que os continha. Nesses casos, quando o nome do contrato é alterado no desenvolvimento, se o nome do construtor também não for alterado, ele se tornará uma função normal e que pode ser chamada. Como você pode imaginar, isso pode levar (e tem) a alguns hacks de contrato interessantes.

Para mais informações, o leitor pode estar interessado em tentar os [desafios](https://github.com/OpenZeppelin/ethernaut) Ethernaut ( em particular o nível Fallout).

A vulnerabilidade

Se o nome do contrato for modificado ou se houver um erro de digitação no nome do construtor que não corresponda ao nome do contrato, o construtor se comportará como uma função normal. Isso pode levar a consequências terríveis, especialmente se o construtor realizar operações privilegiadas. Considere o seguinte contrato:

contrato Proprietário Carteira {

endereço proprietário público;

// construtor

função proprietárioCarteira ( endereço \_proprietário ) public {

dono = \_ dono;

}

// Cair pra trás. Colete éter.

função () a pagar {}

função retirar( ) public {

require( msg.sender == dono);

msg.sender .transfer ( this.balance ) ;

}

}

Este contrato coleta ether e permite que apenas o proprietário o retire, chamando a função de retirada . O problema surge porque o construtor não tem o mesmo nome do contrato: a primeira letra é diferente! Assim, qualquer usuário pode chamar a função ownerWallet , definir-se como o proprietário e, em seguida, obter todo o éter do contrato chamando o método de retirada .

Técnicas Preventivas

Esse problema foi resolvido na versão 0.4.22 do compilador Solidity. Esta versão introduziu uma palavra-chave construtor que especifica o construtor, em vez de exigir que o nome da função corresponda ao nome do contrato. O uso dessa palavra-chave para especificar construtores é recomendado para evitar problemas de nomenclatura.

Exemplo do mundo real: Rubixi

[Rubixi](http://bit.ly/2ESWG7t) foi outro esquema de pirâmide que exibiu esse tipo de vulnerabilidade. Foi originalmente chamado DynamicPyramid , mas o nome do contrato foi alterado antes da implantação para Rubixi . O nome do construtor não foi alterado, permitindo que qualquer usuário se tornasse o criador. Algumas discussões interessantes relacionadas a esse bug podem ser encontradas no [Bitcointalk](http://bit.ly/2P0TRWw) . Em última análise, permitiu que os usuários lutassem pelo status de criador para reivindicar as taxas do esquema de pirâmide. Mais detalhes sobre esse bug específico podem ser encontrados em [“Histórico de vulnerabilidades de segurança do Ethereum, hacks e suas correções”](http://bit.ly/2Q1AMA6) .

Ponteiros de armazenamento não inicializados

O EVM armazena dados como armazenamento ou como memória. Entender exatamente como isso é feito e os tipos padrão para variáveis locais de funções é altamente recomendado ao desenvolver contratos. Isso porque é possível produzir contratos vulneráveis inicializando variáveis de forma inadequada.

Para ler mais sobre armazenamento e memória no EVM, consulte a documentação do Solidity sobre [localização de dados](http://bit.ly/2OdUU0l) , [layout de variáveis de estado no armazenamento](http://bit.ly/2JslDWf) e [layout na memória](http://bit.ly/2Dch2Hc) .

|  |  |
| --- | --- |
| **NOTA** | Esta seção é baseada em um excelente [post de Stefan Beyer](http://bit.ly/2ERI0pb) . Outras leituras sobre este tópico, inspiradas por Stefan, podem ser encontradas neste [tópico do Reddit](http://bit.ly/2OgxPtG) . |

A vulnerabilidade

Variáveis locais dentro de funções são padronizadas para armazenamento ou memória dependendo de seu tipo. As variáveis de armazenamento local não inicializadas podem conter o valor de outras variáveis de armazenamento no contrato; este fato pode causar vulnerabilidades não intencionais, ou ser explorado deliberadamente.

Vamos considerar o contrato de registrador de nomes relativamente simples em [NameRegistrar.sol](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#nameregistrar_security) .

*Exemplo 12. NomeRegistrar.sol*

// Um registrador de nome bloqueado

nome do contratoRegistrar {

bool public desbloqueado = false; // registrador bloqueado, sem atualizações de nome

struct NameRecord { // mapeia hashes para endereços

bytes32 ;

endereço mappedAddress ;

}

// registra quem registrou nomes

mapping( address => NameRecord ) public registradoNameRecord ;

// resolve hashes para endereços

mapping( bytes32 => address) public resolve;

function register( bytes32 \_name, address \_ mappedAddress ) public {

// configura o novo NameRecord

NomeRegistro novoRegistro ;

newRecord.name = \_ nome;

newRecord.mappedAddress = \_mappedAddress ;

resolve[\_name] = \_ mappedAddress ;

RegisterNameRecord [ msg.sender ] = newRecord ;

requer(desbloqueado); // só permite inscrições se o contrato estiver desbloqueado

}

}

Este registrador de nomes simples tem apenas uma função. Quando o contrato é desbloqueado , ele permite que qualquer pessoa registre um nome (como um hash bytes32 ) e mapeie esse nome para um endereço. O registrador é inicialmente bloqueado e a exigência na linha 25 impede que o registrador adicione registros de nome. Parece que o contrato está inutilizável, pois não há como desbloquear o registro! Existe, no entanto, uma vulnerabilidade que permite o registro de nomes independentemente da variável desbloqueada .

Para discutir essa vulnerabilidade, primeiro precisamos entender como o armazenamento funciona no Solidity. Como uma visão geral de alto nível (sem nenhum detalhe técnico adequado - sugerimos a leitura dos documentos do Solidity para uma revisão adequada), as variáveis ​​de estado são armazenadas sequencialmente em *slots* conforme aparecem no contrato (elas podem ser agrupadas, mas não estão neste exemplo, então não vamos nos preocupar com isso). Assim, desbloqueado existe em slot[ 0] , registradoNameRecord em slot[1] , e resolve em slot[2] , etc. Cada um desses slots tem 32 bytes de tamanho (há complexidades adicionais com mapeamentos, que ignoraremos para agora). O booleano desbloqueado será 0x000… ​0 (64 0s, excluindo o 0x ) para false ou 0x000… ​1 (63 0s) para true . Como você pode ver, há um desperdício significativo de armazenamento neste exemplo específico .

A próxima peça do quebra-cabeça é que o Solidity, por padrão, coloca tipos de dados complexos, como structs, no armazenamento ao inicializá-los como variáveis locais. Portanto, newRecord na linha 18 assume o padrão de armazenamento. A vulnerabilidade é causada pelo fato de newRecord não ser inicializado. Como o padrão é storage, ele é mapeado para storage slot[ 0], que atualmente contém um ponteiro para unlocked . Observe que nas linhas 19 e 20 definimos newRecord.name como \_name e newRecord.mappedAddress como \_mappedAddress ; isso atualiza os locais de armazenamento de slot[ 0] e slot[1], que modifica tanto o desbloqueado quanto o slot de armazenamento associado ao RegisterNameRecord .

Isso significa que desbloqueado pode ser modificado diretamente, simplesmente pelo parâmetro bytes32 \_name da função de registro . Portanto, se o último byte de \_name for diferente de zero, ele modificará o último byte do slot de armazenamento [ 0] e mudará diretamente desbloqueado para true . Esses valores \_name farão com que a chamada require na linha 25 seja bem-sucedida, pois definimos unlocked como true . Tente isso no Remix. Observe que a função passará se você usar um \_name do formulário:

0x00000000000000000000000000000000000000000000000000000000000001

Técnicas Preventivas

O compilador Solidity mostra um aviso para variáveis de armazenamento não inicializadas ; os desenvolvedores devem prestar muita atenção a esses avisos ao criar contratos inteligentes. A versão atual do Mist (0.10) não permite que esses contratos sejam compilados. Geralmente, é uma boa prática usar explicitamente os especificadores de memória ou armazenamento ao lidar com tipos complexos, para garantir que eles se comportem conforme o esperado.

Exemplos do mundo real: potes de mel OpenAddressLottery e CryptoRoulette

Um pote de mel chamado [OpenAddressLottery](http://bit.ly/2AAVnWD) foi implantado e usou essa peculiaridade de variável de armazenamento não inicializada para coletar ether de alguns possíveis hackers. O contrato é bastante complicado, então deixaremos a análise para o [tópico do Reddit](http://bit.ly/2OgxPtG) , onde o ataque é explicado com bastante clareza.

Outro pote de mel, [CryptoRoulette](http://bit.ly/2OfNGJ2) , também utilizou esse truque para tentar coletar um pouco de éter. Se você não conseguir descobrir como o ataque funciona, consulte [“An Analysis of a Couple Ethereum Honeypot Contracts”](http://bit.ly/2OVkSL4) para obter uma visão geral deste contrato e de outros.

Ponto flutuante e precisão

No momento da redação deste artigo (v0.4.24), o Solidity não suporta números de ponto fixo e de ponto flutuante. Isso significa que as representações de ponto flutuante devem ser construídas com tipos inteiros no Solidity. Isso pode levar a erros e vulnerabilidades se não for implementado corretamente.

|  |  |
| --- | --- |
| **NOTA** | Para ler mais, consulte o [wiki de dicas e técnicas de segurança de contrato Ethereum](http://bit.ly/2Ogp2Ia) . |

A vulnerabilidade

Como não há tipo de ponto fixo no Solidity, os desenvolvedores precisam implementar seus próprios tipos de dados inteiros padrão. Existem várias armadilhas que os desenvolvedores podem encontrar durante esse processo. Tentaremos destacar alguns deles nesta seção.

Vamos começar com um exemplo de código (ignoraremos os problemas de over/underflow, discutidos anteriormente neste capítulo, para simplificar):

contrato FunWithNumbers {

uint constante public tokensPerEth = 10;

uint constante public weiPerEth = 1e18;

mapping( address => uint ) saldos públicos;

função buyTokens ( ) externo a pagar {

// converte wei em eth, então multiplica pela taxa de token

uint tokens = msg.value / weiPerEth \* tokensPerEth ;

balances[ msg.sender ] += tokens;

}

função sellTokens ( tokens uint ) public {

require(balances[ msg.sender ] >= tokens);

uint eth = tokens/ tokensPerEth ;

balances[ msg.sender ] -= tokens;

msg.sender .transfer ( eth\* weiPerEth );

}

}

Este simples contrato de compra/venda de token tem alguns problemas óbvios. Embora os cálculos matemáticos para comprar e vender tokens estejam corretos, a falta de números de ponto flutuante dará resultados errôneos. Por exemplo, ao comprar tokens na linha 8, se o valor for menor que 1 ether a divisão inicial resultará em 0 , deixando o resultado da multiplicação final como 0 (por exemplo, 200 wei dividido por 1e18 weiPerEth é igual a 0 ). Da mesma forma, ao vender tokens, qualquer número de tokens menor que 10 também resultará em 0 ether . Na verdade, o arredondamento aqui é sempre para baixo, então vender 29 tokens resultará em 2 ether .

O problema com este contrato é que a precisão é apenas para o éter mais próximo (ou seja, 1e18 wei ). Isso pode ser complicado ao lidar com decimais em tokens [ERC20](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md) quando você precisa de maior precisão.

Técnicas Preventivas

Manter a precisão certa em seus contratos inteligentes é muito importante, especialmente ao lidar com índices e taxas que refletem decisões econômicas.

Você deve garantir que quaisquer razões ou taxas que você esteja usando permitam numeradores grandes em frações. Por exemplo, usamos a taxa tokensPerEth em nosso exemplo. Teria sido melhor usar weiPerTokens , que seria um número grande. Para calcular o número correspondente de tokens, poderíamos fazer msg.value / weiPerTokens . Isso daria um resultado mais preciso.

Outra tática a ter em mente é estar atento à ordem das operações. Em nosso exemplo, o cálculo para comprar tokens foi msg.value / weiPerEth \* tokenPerEth . Observe que a divisão ocorre antes da multiplicação. (A solidez, ao contrário de algumas linguagens, garante a execução das operações na ordem em que são escritas.) Este exemplo teria alcançado uma precisão maior se o cálculo realizasse primeiro a multiplicação e depois a divisão; ou seja, msg.value \* tokenPerEth / weiPerEth .

Finalmente, ao definir a precisão arbitrária para números, pode ser uma boa ideia converter valores para uma precisão mais alta, realizar todas as operações matemáticas e, finalmente, converter novamente para a precisão necessária para a saída. Normalmente são usados uint256s (pois são ideais para uso de gás); estes dão aproximadamente 60 ordens de grandeza em seu alcance, alguns dos quais podem ser dedicados à precisão de operações matemáticas. Pode ser que seja melhor manter todas as variáveis em alta precisão no Solidity e convertê-las novamente em precisões mais baixas em aplicativos externos (é essencialmente assim que a variável decimals funciona em contratos de token ERC20). Para ver um exemplo de como isso pode ser feito, recomendamos consultar [DS-Math](https://github.com/dapphub/ds-math) . Ele usa alguns nomes estranhos (“wads” e “rays”), mas o conceito é útil.

Exemplo do mundo real: Ethstick

O [contrato](http://bit.ly/2Qb7PSB) Ethstick não usa precisão estendida; no entanto, ele lida com wei . Então, esse contrato terá problemas de arredondamento, mas apenas no nível de precisão wei . Ele tem algumas falhas mais sérias, mas elas estão relacionadas à dificuldade de obter entropia no blockchain (consulte [Entropy Illusion](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#entropyillusion_security) ). Para uma discussão mais aprofundada do contrato Ethstick , vamos encaminhá-lo para outro post de Peter Vessenes , [“Os contratos Ethereum vão ser doces para hackers”](http://bit.ly/2SwDnE0) .

Autenticação Tx.Origin

Solidity tem uma variável global, tx.origin , que percorre toda a pilha de chamadas e contém o endereço da conta que originalmente enviou a chamada (ou transação). Usar essa variável para autenticação em um contrato inteligente deixa o contrato vulnerável a um ataque de phishing.

|  |  |
| --- | --- |
| **NOTA** | Para ler mais, veja a pergunta Ethereum [Stack Exchange de dbryson](http://bit.ly/2PxU1UM) , [“ Tx.Origin and Ethereum Oh My!”](http://bit.ly/2qm7ocJ) por Peter Vessenes , e [“Solidity: Tx Origin Attacks”](http://bit.ly/2P3KVA4) por Chris Coverdale. |

A vulnerabilidade

Contratos que autorizam usuários usando a variável tx.origin normalmente são vulneráveis a ataques de phishing que podem induzir os usuários a realizar ações autenticadas no contrato vulnerável. Considere o contrato simples em [Phishable.sol](file:///\\wsl.localhost\Ubuntu-20.04\home\syr3fx\Code\github\maitriser-ca\LivreMaitriserEthereum\adoc\for_use_ethereumbook_09smart-contracts-securitytxt_fr_CA.html#phishable_security) .

*Exemplo 13. Phishable.sol*

contrato Phishable {

endereço proprietário público;

construtor (endereço \_proprietário) {

dono = \_ dono;

}

function () externo a pagar {} // coleta ether

função retireAll ( endereço \_recipient) public {

require( tx.origin == dono);

\_ destinatário.transferência ( this.balance );

}

}

Observe que na linha 11 o contrato autoriza a função retirarAll usando tx.origin . Este contrato permite que um invasor crie um contrato de ataque da forma:

import " Phishable.sol ";

contrato AtaqueContrato {

Phishable phishingContract ;

atacante de endereço ; // O endereço do invasor para receber fundos

construtor ( Phishable \_ phishableContract , address \_ attackerAddress ) {

phishableContract = \_ phishableContract ;

atacante = \_ endereço do atacante ;

}

função () a pagar {

phishableContract.withdrawAll (atacante );

}

}

O invasor pode disfarçar esse contrato como seu próprio endereço privado e projetar socialmente a vítima (o proprietário do contrato Phishable ) para enviar alguma forma de transação para o endereço - talvez enviando a esse contrato alguma quantidade de éter. A vítima, a menos que seja cuidadosa, pode não perceber que há código no endereço do invasor, ou o invasor pode passar por uma carteira multiassinatura ou alguma carteira de armazenamento avançado (lembre-se de que o código-fonte dos contratos públicos não está disponível por padrão) .

De qualquer forma, se a vítima enviar uma transação com gás suficiente para o endereço AttackContract , ela invocará a função fallback, que por sua vez chama a função retirarAll do contrato Phishable com o parâmetro atacante . Isso resultará na retirada de todos os fundos do contrato Phishable para o endereço do invasor . Isso ocorre porque o primeiro endereço que inicializou a chamada foi a vítima (ou seja, o proprietário do contrato Phishable ). Portanto, tx.origin será igual ao proprietário e a exigência na linha 11 do contrato Phishable passará.

Técnicas Preventivas

tx.origin não deve ser usado para autorização em contratos inteligentes. Isso não quer dizer que a variável tx.origin nunca deva ser usada. Ele tem alguns casos de uso legítimos em contratos inteligentes. Por exemplo, se alguém quiser impedir que contratos externos chamem o contrato atual, pode-se implementar um require no formato require( tx.origin == msg.sender ) . Isso evita que contratos intermediários sejam usados para chamar o contrato atual, limitando o contrato a endereços regulares sem código.

Bibliotecas de contrato

Há muito código existente disponível para reutilização, tanto implantado na cadeia como bibliotecas que podem ser chamadas e fora da cadeia como bibliotecas de modelo de código. As bibliotecas na plataforma, depois de implantadas, existem como contratos inteligentes de bytecode, portanto, muito cuidado deve ser tomado antes de usá-las na produção. No entanto, o uso de bibliotecas na plataforma existentes bem estabelecidas traz muitas vantagens, como poder se beneficiar das atualizações mais recentes, economizar dinheiro e beneficiar o ecossistema Ethereum, reduzindo o número total de contratos ativos no Ethereum.

No Ethereum, o recurso mais utilizado é a [suíte](https://openzeppelin.org/) OpenZeppelin , uma ampla biblioteca de contratos que vão desde implementações de tokens ERC20 e ERC721, a muitos tipos de modelos de crowdsale , a comportamentos simples comumente encontrados em contratos, como Ownable , Pausable ou LimitBalance . Os contratos neste repositório foram amplamente testados e, em alguns casos, até funcionam como implementações padrão *de fato .* Eles são de uso gratuito e são construídos e mantidos pelo [Zeppelin](https://zeppelin.solutions) juntamente com uma lista cada vez maior de contribuidores externos.

Também do Zeppelin é o [ZeppelinOS](https://zeppelinos.org/) , uma plataforma de código aberto de serviços e ferramentas para desenvolver e gerenciar aplicativos de contrato inteligente com segurança. O ZeppelinOS fornece uma camada no topo do EVM que facilita para os desenvolvedores lançarem DApps atualizáveis vinculados a uma biblioteca on-chain de contratos bem testados que podem ser atualizados. Diferentes versões dessas bibliotecas podem coexistir na plataforma Ethereum, e um sistema de garantia permite que os usuários proponham ou impulsionem melhorias em diferentes direções. Um conjunto de ferramentas off-chain para depurar, testar, implantar e monitorar aplicativos descentralizados também é fornecido pela plataforma.

O projeto ethpm visa organizar os diversos recursos que estão se desenvolvendo no ecossistema fornecendo um sistema de gerenciamento de pacotes. Como tal, seu registro fornece mais exemplos para você navegar:

* Site: <https://www.ethpm.com/>
* Link do repositório : <https://www.ethpm.com/registry>
* Link do GitHub: <https://github.com/ethpm>
* Documentação: <https://www.ethpm.com/docs/integration-guide>

Conclusões

Há muito para qualquer desenvolvedor que trabalha no domínio do contrato inteligente saber e entender. Ao seguir as melhores práticas em seu design de contrato inteligente e escrita de código, você evitará muitas armadilhas e armadilhas graves.

Talvez o princípio de segurança de software mais fundamental seja maximizar a reutilização de código confiável. Em criptografia, isso é tão importante que foi condensado em um ditado: "Não role sua própria criptografia". No caso de contratos inteligentes, isso equivale a ganhar o máximo possível de bibliotecas disponíveis gratuitamente que foram minuciosamente examinadas pela comunidade.