Aula 14 – Escrevendo Testes Automatizados para Smart Contracts

No universo da tecnologia, especialmente no campo dinâmico e de alto risco dos smart contracts, a confiança é a moeda mais valiosa. Imagine construir um arranha-céu sem testar a resistência de seus pilares, ou lançar um foguete sem simular cada etapa da decolagem. No mundo dos contratos inteligentes, onde o código é lei e as transações são irreversíveis, a ausência de testes rigorosos pode ter consequências catastróficas, resultando em perdas financeiras massivas e na erosão da credibilidade.

Esta aula foi cuidadosamente desenhada para você, que busca não apenas entender, mas dominar a arte de construir sistemas robustos e seguros no ecossistema blockchain. Nosso foco será desvendar a importância crítica dos testes automatizados e equipá-lo com as ferramentas e o conhecimento necessários para garantir a integridade de seus smart contracts. Ao final desta jornada, você será capaz de aplicar as melhores práticas de teste, utilizando frameworks amplamente adotados pela indústria, como Hardhat, Mocha e Chai, para proteger seus projetos e, por extensão, os ativos digitais que eles gerenciam.

Vamos explorar juntos como a segurança se tornou a prioridade máxima no desenvolvimento Web3, e como a incorporação de testes automatizados desde as fases iniciais do projeto não é apenas uma boa prática, mas uma necessidade inegociável. Prepare-se para mergulhar em um tópico que transformará sua abordagem ao desenvolvimento de smart contracts, elevando seu trabalho a um novo patamar de profissionalismo e confiabilidade.

A Importância Vital dos Testes para a Segurança dos Smart Contracts

Pense nos smart contracts como acordos digitais autônomos, que vivem em uma blockchain e executam suas cláusulas sem a necessidade de intermediários. Uma vez implantados, eles são, em sua maioria, imutáveis. Essa imutabilidade, que é uma das maiores forças da tecnologia blockchain, também se torna sua maior vulnerabilidade se o código contiver falhas. Um erro, por menor que seja, pode ser explorado por agentes mal-intencionados, levando a perdas irrecuperáveis de fundos ou à paralisação de sistemas inteiros.

> **Casos históricos:** O ataque ao DAO resultou na perda de milhões de dólares e na divisão da rede Ethereum. As vulnerabilidades em contratos da Parity Wallet congelaram centenas de milhões em Ether.

É aqui que os testes automatizados entram como uma camada essencial de proteção. Eles agem como um exército de inspetores incansáveis, verificando cada linha de código, cada lógica de negócio, cada interação possível, antes que o contrato seja exposto ao mundo real. Ao identificar e corrigir falhas em um ambiente controlado, os testes nos permitem construir com confiança, sabendo que estamos minimizando os riscos inerentes a essa tecnologia revolucionária.

O Custo de um Erro: Por Que Não Podemos Falhar

Software Tradicional

- Bugs podem ser corrigidos com atualizações
- Patches podem ser aplicados rapidamente
- Rollback é possível
- Impacto geralmente limitado

Smart Contracts

- **Imutabilidade permanente**
- Falhas se tornam características
- Migração complexa e arriscada
- Perdas financeiras irreversíveis

Imagine que você está construindo um cofre bancário digital, onde bilhões de dólares serão armazenados. Cada porta, cada mecanismo de travamento, cada sensor precisa funcionar perfeitamente. Se houver uma falha na lógica de acesso, todo o dinheiro pode ser roubado sem deixar rastros, e o cofre não pode ser simplesmente "reparado" no local. A única opção seria construir um novo cofre e transferir os fundos, se ainda houver algum.

> *"A confiança, uma vez quebrada, é extremamente difícil de reconstruir. Por isso, a mentalidade de 'não podemos falhar' deve permear todo o processo de desenvolvimento de smart contracts."*

Essa analogia ilustra a gravidade. Um erro em um smart contract pode levar não apenas a perdas financeiras diretas, mas também a danos irreparáveis à reputação do projeto e dos desenvolvedores. Os testes automatizados são a nossa garantia de que o cofre digital que estamos construindo é, de fato, impenetrável.

Tipos de Testes em Smart Contracts: Uma Visão Geral

Quando falamos em "testar", é importante entender que não existe uma solução única para todas as necessidades. Assim como um médico utiliza diferentes exames para diagnosticar um paciente, nós, desenvolvedores de smart contracts, empregamos uma variedade de técnicas de teste para garantir a saúde e a segurança do nosso código. Cada tipo de teste atua em uma camada diferente, oferecendo uma perspectiva única sobre o comportamento e a robustez do contrato.

Testes Unitários

Verificam a menor unidade de código possível – uma função específica – de forma isolada. Como testar cada peça de um motor individualmente.

Testes de Integração

Verificam como diferentes contratos ou módulos interagem entre si, garantindo que as peças se encaixem perfeitamente.

Testes End-to-End

Simulam o fluxo completo do usuário, desde a interface até a blockchain, testando toda a jornada.

Fuzzing

Injeta dados aleatórios para encontrar comportamentos inesperados e vulnerabilidades ocultas.

Verificação Formal

Usa lógica matemática para provar a correção do código de forma absoluta.

Foco desta aula: Embora esta aula se concentre nos testes unitários, é crucial ter em mente que uma estratégia de teste abrangente envolve múltiplas camadas de validação para construir sistemas verdadeiramente resilientes.

Entendendo o Ambiente de Testes: Hardhat no Centro

Para escrever testes eficazes, precisamos de um ambiente que nos permita simular a blockchain de forma controlada e eficiente. É aqui que o Hardhat entra em cena, não apenas como uma ferramenta, mas como um ecossistema completo de desenvolvimento para smart contracts. Ele oferece uma rede Ethereum local para testes, ferramentas de depuração e, crucialmente para esta aula, um framework robusto para a execução de testes automatizados.

01

Simulação Local

Cria uma blockchain Ethereum local para testes rápidos e sem custos

02

Ferramentas de Debug

Oferece recursos avançados para identificar e corrigir problemas

03

Framework de Testes

Integra-se perfeitamente com Mocha e Chai para testes robustos

04

Ecossistema de Plugins

Extensível com plugins que adicionam funcionalidades específicas

Pense no Hardhat como uma bancada de testes de alta tecnologia para engenheiros. Em vez de construir e testar um protótipo em condições reais, o que seria caro e demorado, você pode simular todas as condições e interações em um ambiente controlado. Isso permite que você itere rapidamente, identifique problemas e refine seu código sem o custo e a lentidão de uma blockchain pública.

A escolha do Hardhat como nossa ferramenta principal não é por acaso. Ele é amplamente adotado pela indústria, conhecido por sua flexibilidade e pela riqueza de plugins que estendem suas funcionalidades. Ao dominar o Hardhat, você estará não apenas aprendendo a testar, mas também a trabalhar com uma das plataformas de desenvolvimento Web3 mais relevantes e atualizadas do mercado, preparando-o para os desafios do mundo real.

Mocha: O Framework de Testes que Organiza o Caos

Com o Hardhat configurado como nosso ambiente, o próximo passo é introduzir o Mocha, um framework de testes JavaScript que nos ajuda a estruturar e executar nossos testes de forma organizada. Se o Hardhat é a bancada de testes, o Mocha é o manual de instruções que nos diz como organizar e conduzir cada experimento. Ele não faz as verificações em si, mas fornece a estrutura para descrever o que estamos testando e como.

Estrutura Clara

Como um livro de receitas com capítulos (describe) e seções (it) para cada receita individual

Agrupamento Lógico

Permite agrupar testes relacionados, facilitando a navegação e manutenção

Ganchos de Configuração

beforeEach() e afterEach() garantem ambiente limpo para cada teste

Funções Principais do Mocha

describe()

Agrupa testes relacionados em blocos lógicos, criando uma hierarquia clara

it()

Define um caso de teste individual com uma descrição clara do comportamento esperado

beforeEach()

Configura o estado antes de cada teste, garantindo isolamento e previsibilidade

afterEach()

Limpa o ambiente após cada teste, mantendo a independência entre testes

A beleza do Mocha reside em sua simplicidade e flexibilidade. Essa estrutura é fundamental para escrever testes que sejam robustos e fáceis de depurar, especialmente à medida que seu projeto cresce em complexidade.

Chai: A Linguagem da Afirmação nos Seus Testes

Se o Mocha nos dá a estrutura para organizar nossos testes, o Chai nos fornece a linguagem para expressar as expectativas. Depois de executar uma função do nosso smart contract em um teste, precisamos verificar se o resultado é o que esperávamos. É aqui que o Chai, uma biblioteca de asserções, entra em jogo.

O Inspetor de Qualidade

Pense no Chai como um inspetor de qualidade rigoroso. Depois que você produz um item (o resultado da sua função), o inspetor Chai verifica se ele atende a todas as especificações.

- **Eu espero que este item seja igual a este**
- **Eu espero que este item contenha aquilo**
- **Eu espero que este item não seja nulo**

Estilo Expect: Asserções Legíveis

```
expect(await contract.balanceOf(owner.address)).to.equal(1000);
```

O Chai oferece diferentes estilos de asserção, mas o mais popular e legível para testes de smart contracts é o estilo **expect**. Essa frase, quase como uma sentença em inglês, deixa claro o que o teste está verificando.

Clareza

Asserções que se leem como linguagem natural, facilitando a compreensão

Flexibilidade

Múltiplos matchers para diferentes tipos de verificações

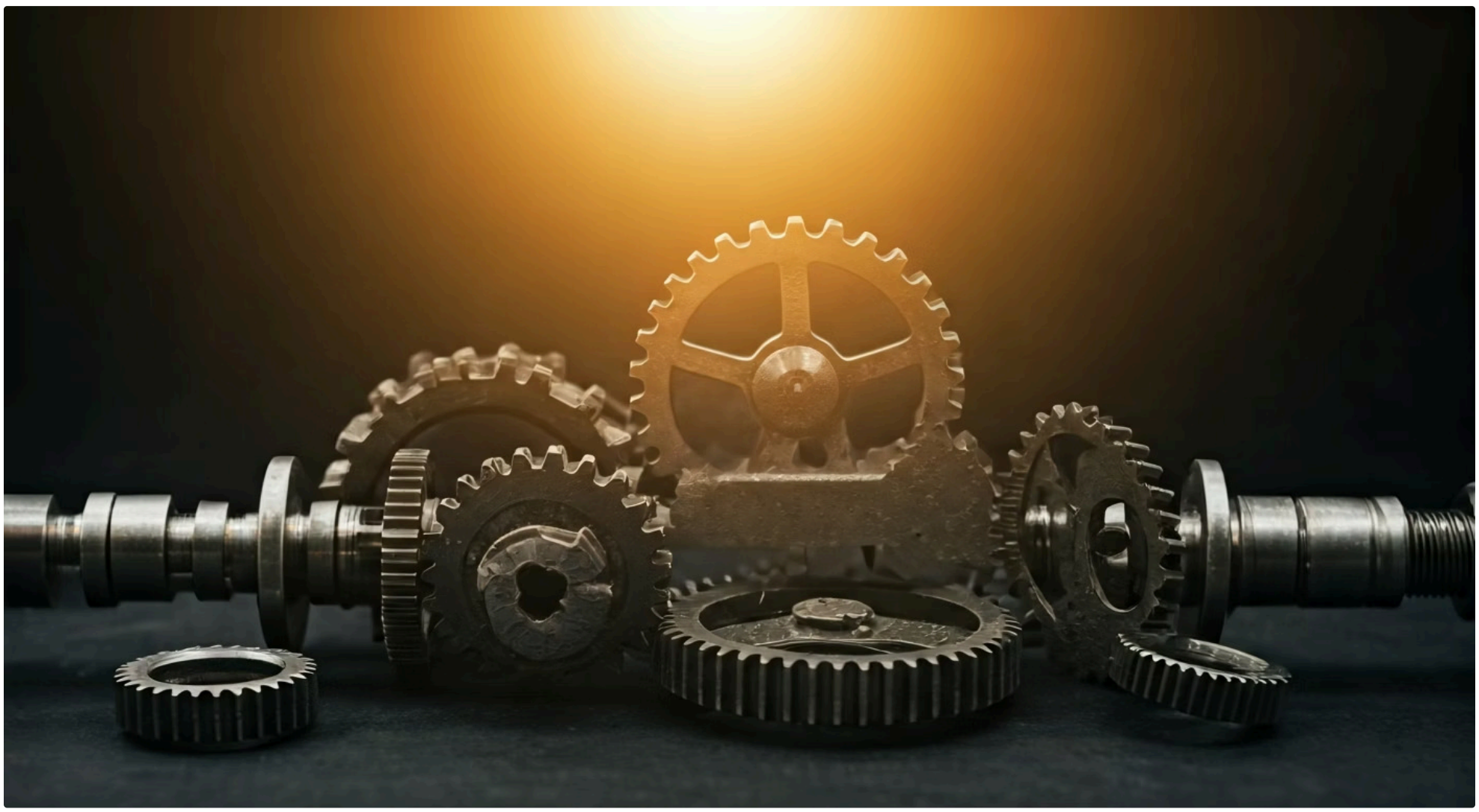
Documentação Viva

Testes se tornam documentação do comportamento esperado

Ao usar o Chai, você está transformando as verificações técnicas em declarações compreensíveis, tornando seus testes uma documentação viva do comportamento esperado do seu contrato.

Ferramentas Integradas

A Sinergia entre Hardhat, Mocha e Chai



Cada ferramenta desempenha um papel específico e essencial no ecossistema de testes, criando uma sinergia poderosa que permite construir testes robustos e confiáveis para seus smart contracts.

Configurando o Hardhat para Testes Unitários

Agora que entendemos os papéis do Hardhat, Mocha e Chai, é hora de colocá-los para trabalhar. A configuração inicial é um passo crucial que estabelece a base para todo o seu ambiente de testes. Felizmente, o Hardhat simplifica bastante esse processo, permitindo que você comece a escrever testes rapidamente.

Inicializar Projeto

Crie um novo projeto Hardhat com `npx hardhat`

Instalar Dependências

Instale Mocha, Chai e ferramentas necessárias via npm

Configurar hardhat.config.js

Defina redes, compiladores e configurações de teste

Pronto para Testar

Ambiente configurado e pronto para escrever testes

Instalação das Dependências

```
npm install --save-dev hardhat @nomicfoundation/hardhat-toolbox @nomicfoundation/hardhat-chai-matchers chai
```

Configuração Básica do hardhat.config.js

```
// hardhat.config.js
require("@nomicfoundation/hardhat-toolbox");

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
  solidity: "0.8.20", // Versão do compilador Solidity
  networks: {
    hardhat: { // Configurações da rede local do Hardhat para testes
      chainId: 31337, // ID de rede padrão do Hardhat
    },
  },
  paths: {
    sources: "./contracts",
    tests: "./test", // Onde seus arquivos de teste serão armazenados
    cache: "./cache",
    artifacts: "./artifacts",
  },
  mocha: {
    timeout: 40000, // Aumenta o timeout para testes mais complexos
  },
};
```

Pronto para começar: Com essa configuração básica, você está pronto para criar seu primeiro arquivo de teste na pasta ./test e começar a escrever as asserções que garantirão a segurança e a correção do seu smart contract.

Escrevendo Seu Primeiro Teste Unitário com Hardhat, Mocha e Chai

Chegou a hora de colocar a mão na massa e escrever nosso primeiro teste. Vamos imaginar que temos um contrato simples chamado **MyToken** que herda de ERC20 da OpenZeppelin e tem um totalSupply fixo. Nosso objetivo é testar se o contrato é implantado corretamente e se a quantidade total de tokens é a esperada.

Passo 1: Criar o Contrato MyToken.sol

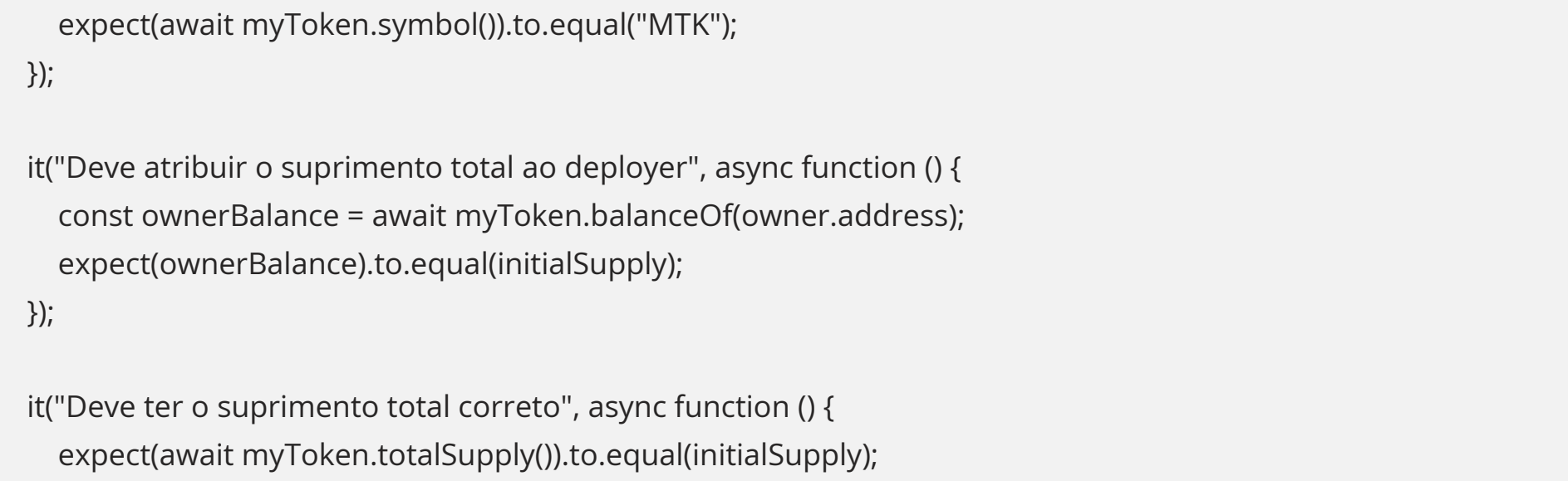
```
// contracts/MyToken.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("MyToken", "MTK") {
        _mint(msg.sender, initialSupply);
    }
}
```

Passo 2: Criar o Arquivo de Teste MyToken.test.js

```
// test/MyToken.test.js
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("MyToken", function () {
    let MyToken;
    let myToken;
    let owner;
    let addr1;
    let addr2;
    let initialSupply = ethers.parseEther("1000"); // 1000 tokens

    beforeEach(async function () {
        // Obtém as contas de teste do Hardhat
        [owner, addr1, addr2] = await ethers.getSigners();

        // Implanta o contrato MyToken
        MyToken = await ethers.getContractFactory("MyToken");
        myToken = await MyToken.deploy(initialSupply);
        await myToken.waitForDeployment(); // Espera a implantação ser confirmada
    });

    it("Deve ter o nome e símbolo corretos", async function () {
        expect(await myToken.name()).to.equal("MyToken");
        expect(await myToken.symbol()).to.equal("MTK");
    });

    it("Deve atribuir o suprimento total ao deployer", async function () {
        const ownerBalance = await myToken.balanceOf(owner.address);
        expect(ownerBalance).to.equal(initialSupply);
    });

    it("Deve ter o suprimento total correto", async function () {
        expect(await myToken.totalSupply()).to.equal(initialSupply);
    });
});
```

Passo 3: Executar os Testes

```
npx hardhat test
```



Testes Escritos
Verificando nome, símbolo e suprimento

Cobertura Básica
Funcionalidades essenciais testadas

Comando
Para executar todos os testes

Parabéns! Este é o seu primeiro passo para garantir a segurança e a funcionalidade do seu smart contract através de testes automatizados.

Testando Funções de Escrita: Alterando o Estado do Contrato

Até agora, testamos funções que apenas leem o estado do contrato (como name(), symbol(), balanceOf(), totalSupply()). Mas a maioria dos smart contracts envolve funções que modificam o estado da blockchain, como transferências de tokens, aprovações ou interações com outros contratos. Testar essas funções de escrita é crucial para garantir que elas se comportem como esperado e que as mudanças de estado ocorram corretamente.

Analogia: Imagine que você está testando um sistema de votação. Não basta verificar se o nome do candidato está correto; você precisa garantir que, ao votar, o contador de votos do candidato seja incrementado e que o eleitor não possa votar novamente.

Requisitos para Testar Funções de Escrita

Conectar Diferentes Contas

Use contract.connect(signer) para simular chamadas de diferentes usuários

Aguardar Transações

Use await transaction.wait() para garantir que a transação foi minerada

Exemplo: Testando a Função transfer()

```
// test/MyToken.test.js (continuação)
// ... dentro do bloco describe("MyToken", function () { ...

it("Deve permitir a transferência de tokens entre contas", async function () {
    // Transferir 50 tokens do owner para addr1
    const transferAmount = ethers.parseEther("50");
    await myToken.transfer(addr1.address, transferAmount);

    // Verificar o saldo de addr1
    const addr1Balance = await myToken.balanceOf(addr1.address);
    expect(addr1Balance).to.equal(transferAmount);

    // Verificar o saldo do owner (deve ter diminuído)
    const ownerBalance = await myToken.balanceOf(owner.address);
    expect(ownerBalance).to.equal(initialSupply - transferAmount);
});

it("Deve falhar se o remetente não tiver saldo suficiente", async function () {
    const transferAmount = ethers.parseEther("2000"); // Mais do que o owner possui

    // Usamos .to.be.revertedWith para verificar se a transação falha com uma mensagem específica
    await expect(myToken.transfer(addr1.address, transferAmount))
        .to.be.revertedWith("ERC20: transfer amount exceeds balance");
});
```

Este exemplo demonstra como testar tanto o sucesso de uma transferência quanto a falha esperada quando as condições não são atendidas, garantindo que seu contrato lide com diferentes cenários de forma robusta.

Lidando com Reverts e Exceções: Testes de Falha

Por Que Testar Falhas?

Um contrato inteligente robusto não é apenas aquele que executa suas funções corretamente, mas também aquele que **falha** de forma controlada e previsível quando as condições não são atendidas. Testar esses cenários de falha é tão importante quanto testar os cenários de sucesso.

Analogia do Caixa Eletrônico

Imagine um caixa eletrônico. Ele deve permitir saques quando há saldo suficiente, mas é igualmente crucial que ele *recuse* um saque se:

- O saldo for insuficiente
- O cartão estiver bloqueado
- O limite diário foi atingido
- Há suspeita de fraude

Testando Reverts com Chai

Com o Chai e o Hardhat, testar reverts é direto. A biblioteca `@nomicfoundation/hardhat-chai-matchers` estende o Chai com matchers específicos para transações de blockchain, como `to.be.revertedWith()`.

to.be.revertedWith()

Verifica se a transação falha com uma mensagem de erro específica

to.be.reverted

Verifica se a transação falha, independente da mensagem

to.be.revertedWithCustomError()

Verifica se a transação falha com um erro customizado específico

Exemplo Prático

```
// test/MyToken.test.js (continuação)
// ... dentro do bloco describe("MyToken", function () { ...

it("Deve falhar se um usuário sem permissão tentar chamar uma função restrita", async function () {
    const transferAmount = ethers.parseEther("100");

    // Conectando addr1 para tentar transferir de addr1, que não tem tokens ainda.
    await expect(myToken.connect(addr1).transfer(addr2.address, transferAmount))
        .to.be.revertedWith("ERC20: transfer amount exceeds balance");
});
```

> "Testar esses 'caminhos de falha' é uma prática essencial para construir contratos inteligentes verdadeiramente robustos e seguros, antecipando e mitigando potenciais vetores de ataque."

Organizando Seus Testes: Boas Práticas e Padrões

À medida que seus smart contracts crescem em complexidade, sua suíte de testes também crescerá. Sem uma boa organização, os testes podem se tornar difíceis de ler, manter e depurar, anulando muitos dos benefícios de tê-los. Boas práticas de organização são como a arquitetura de um edifício: elas garantem que a estrutura seja sólida, escalável e fácil de navegar.

Hierarquia Lógica

Use describe() aninhados para criar uma estrutura que reflete a organização do seu contrato ou funcionalidades

Nomenclatura Clara

Nomes de testes devem descrever o que está sendo testado e qual é o resultado esperado

Isolamento de Testes

Use beforeEach() para garantir que cada teste seja independente e não afetado por testes anteriores

Quadro Comparativo: Boas Práticas de Organização de Testes

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
describe()	Agrupamento lógico de testes	Mocha	describe("MyToken", function() { ... });
it()	Definição de um caso de teste individual	Mocha	it("Deve transferir tokens", async function() { ... });
beforeEach()	Configuração de estado antes de cada teste	Mocha	Implantação do contrato, atribuição de contas antes de cada it.
Nomenclatura Clara	Legibilidade e manutenibilidade	Padrão de desenvolvimento de software	"Deve reverter se o remetente não tiver saldo" vs. "Teste de erro".

Dica profissional: Um bom nome de teste deve descrever *o que* está sendo testado e *qual* é o resultado esperado. Isso torna seus testes uma documentação viva do comportamento do contrato.

Integrando OpenZeppelin: Testando Contratos Padrão

A OpenZeppelin é uma biblioteca de contratos inteligentes auditados e seguros, que fornece implementações padronizadas de funcionalidades comuns, como tokens ERC20, ERC721 e mecanismos de controle de acesso. Utilizar a OpenZeppelin é uma prática recomendada para aumentar a segurança e a confiabilidade dos seus contratos, pois você está construindo sobre uma base testada e aprovada pela comunidade.

O Que Testar ao Usar OpenZeppelin?

No entanto, o fato de você usar contratos da OpenZeppelin não significa que você não precisa testar. Pelo contrário! Você não precisa testar o código *da OpenZeppelin* (eles já fazem isso extensivamente), mas precisa testar **como o seu contrato interage** com eles e **qualquer lógica personalizada** que você adicione.

01

Inicialização

Garantir que o construtor do seu contrato chame corretamente o construtor do contrato pai da OpenZeppelin

02

Lógica Personalizada

Testar qualquer função ou modificador que você adicione ao contrato base da OpenZeppelin

03

Interações

Se o seu contrato interage com outros contratos OpenZeppelin, teste essas interações

> *"Pense nisso como construir uma casa usando tijolos certificados. Você não precisa testar a qualidade de cada tijolo, mas precisa testar a integridade da parede que você construiu com eles."*

O exemplo do MyToken que vimos anteriormente já ilustra essa integração, pois ele herda de ERC20 da OpenZeppelin. Nossos testes verificaram o name, symbol, totalSupply e transfer, que são funcionalidades do ERC20, garantindo que a herança e a inicialização funcionaram como esperado. Ao focar na sua lógica personalizada e nas interações, você garante que a robustez da OpenZeppelin se estenda ao seu projeto.

Além dos Testes Unitários: Uma Breve Visão

Embora os testes unitários sejam a espinha dorsal de uma estratégia de segurança robusta para smart contracts, eles são apenas uma peça do quebra-cabeça. Para construir sistemas verdadeiramente resilientes, é essencial adotar uma abordagem de segurança em camadas, onde diferentes tipos de testes e verificações trabalham em conjunto para cobrir o máximo de cenários possível.

> **Analogia:** Imagine a segurança de um aeroporto. Não basta apenas inspecionar a bagagem (teste unitário); é preciso verificar a identidade dos passageiros (teste de integração), monitorar o tráfego aéreo (teste end-to-end), e ter planos de contingência para emergências (verificação formal e fuzzing).

Camadas de Segurança em Testes

Testes Unitários

Base fundamental - testam funções individuais isoladamente

Testes de Integração

Verificam como múltiplos contratos interagem entre si

Testes End-to-End

Simulam a experiência completa do usuário, da interface à blockchain

Fuzzing

Injeta entradas aleatórias para descobrir comportamentos não previstos

Verificação Formal

Prova matemática da correção de propriedades do contrato

A combinação dessas abordagens cria uma rede de segurança abrangente, minimizando os riscos e aumentando a confiança nos seus smart contracts. Os testes unitários são o ponto de partida, mas a jornada para a segurança completa é contínua e multifacetada.

Consolidação e Próximos Passos

Sua Jornada na Segurança de Smart Contracts

Chegamos ao final de nossa jornada sobre testes automatizados para smart contracts. Vimos que, no mundo da Web3, onde o código é lei e a imutabilidade é a norma, a importância de testar rigorosamente não pode ser subestimada. Compreendemos que ferramentas como Hardhat, Mocha e Chai são essenciais para construir uma base sólida de segurança e confiança em nossos projetos.

Mentalidade de Segurança

Adote uma abordagem proativa, escrevendo testes unitários para cada função do seu contrato

Organização Clara

Use describe e it para organizar seus testes de forma lógica e expect do Chai para asserções claras

Teste Completo

Não se esqueça de testar tanto os caminhos de sucesso quanto os de falha

Autoavaliação

1

Qual é a principal razão pela qual os testes automatizados são considerados cruciais para a segurança de smart contracts?

- a) Aumentar a velocidade de implantação dos contratos.
- b) Reduzir o custo de gás das transações.
- c) Garantir a imutabilidade do código após a implantação, prevenindo falhas irreversíveis.
- d) Simplificar a interface do usuário para interagir com o contrato.

2

No contexto de testes com Hardhat, qual é a função principal do Mocha?

- a) Fornecer uma rede Ethereum local para execução dos testes.
- b) Definir a linguagem de asserção para verificar os resultados dos testes.
- c) Estruturar e organizar os testes em blocos lógicos (describe, it).
- d) Compilar o código Solidity antes da execução dos testes.

3

Ao testar uma função de smart contract que deve falhar sob certas condições (ex: saldo insuficiente), qual matcher do Chai é mais apropriado para verificar essa falha e sua mensagem?

- a) to.equal()
- b) to.be.true()
- c) to.be.revertedWith()
- d) to.exist()

4

Qual das seguintes afirmações sobre a integração de contratos OpenZeppelin em seus testes é verdadeira?

- a) Não é necessário testar contratos que herdam da OpenZeppelin, pois eles já são auditados.
- b) Deve-se focar em testar a lógica personalizada adicionada ao contrato base da OpenZeppelin e suas interações.
- c) A OpenZeppelin fornece seu próprio framework de testes, que deve ser usado em vez de Hardhat.
- d) Contratos OpenZeppelin não podem ser testados com Hardhat, Mocha e Chai.

5

Explique a importância de testar os "caminhos de falha" (cenários onde o contrato deve reverter) em smart contracts e como isso contribui para a robustez e segurança do sistema.

Gabarito

1

Resposta: **c)**

2

Resposta: **c)**

3

Resposta: **c)**

4

Resposta: **b)**

Próxima Aula

Aula 15: Scripts de Deploy - Publicando em Redes de Teste

Você aprenderá a automatizar o processo de implantação de seus smart contracts, movendo-os do ambiente de desenvolvimento local para redes de teste públicas, preparando-os para o lançamento em produção.

Recursos Adicionais

Documentação Oficial do Hardhat

Para aprofundar-se nas funcionalidades e plugins do framework

Documentação do Mocha

Para explorar todas as opções de organização e ganchos de teste

Documentação do Chai

Para descobrir a gama completa de asserções disponíveis

OpenZeppelin Contracts

Para entender os padrões e implementações de contratos seguros

NOTA IMPORTANTE: As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.