

Лабораторная работа 4. Композируемость смарт-контрактов

Цель

Познакомиться на практике с композируемостью смарт-контрактов. Научиться вызывать функции смарт-контракта из другого контракта.

Шаги:

1. Модифицировать смарт-контракт Poster
2. Развернуть его на блокчейне
3. Создать пользовательский интерфейс для взаимодействия со смарт-контрактом

Замечание о связи с предыдущими работами

Данная лабораторная базируется на результатах работ 2 и 3. Рекомендуется выполнить лабораторные работы 2 и 3 прежде чем приступить к текущей.

Замечание об окружении

Ниже подразумевается, что вы работаете в UNIX-подобной системе: Linux, FreeBSD, MacOS, и т.п. Если вы работаете с Microsoft Windows, вам потребуется дополнительно установить [Windows Subsystem for Linux](#) или использовать виртуальную машину с Linux через [Vagrant](#), [VirtualBox](#) и др.

Модифицировать смарт-контракт Poster

Композируемость - свойство информационных систем, означающее, что предоставляемые компоненты можно свободно сочетать друг с другом. В данной лабораторной работе мы рассматриваем композируемость смарт-контрактов: возможность вызывать функции одного смарт-контракта из другого контракта.

В лабораторной работе 2 вы создавали смарт-контракт Poster. Он позволяет любому человеку записывать текстовые сообщения в контракт. В данной работе мы модифицируем контракт так, чтобы записывать сообщения могли только владельцы токенов с балансом, большим заданного.

В конце работы 2 у вас должна была получиться папка `poster-contract`, содержащая контракт Poster в файле `contracts/Poster.sol` и миграцию `migrations/2_poster.js`. Скопируйте содержимое папки `poster-contract` в папку `token-gated-poster`. В нашей работе мы будем использовать эту папку `token-gated-poster`, фактически копию результата работы 2.

По задаче нам необходимо сделать так, чтобы только владельцы заданного токена с балансом больше заданного числа могли создавать сообщения. Нам нужно вызвать функцию `balanceOf` этого токена и проверить, больше ли это заданного порога. Отсюда мы можем вывести, что в контракте Poster нам необходимо иметь:

- адрес токена,
- пороговое значение.

Зададим эти величины как переменные контракта. Добавим их в контракт и инициализируем их из параметров конструктора:

```
// ... предыдущий код
contract Poster {
    address tokenAddress;
    uint256 threshold;

    constructor(address _tokenAddress, uint256 _threshold) {
        tokenAddress = _tokenAddress;
        threshold = _threshold;
    }
}
/// ... следующий код
```

Для того, чтобы вызывать функцию контракта необходимо знать его интерфейс. Наш токен из работы 3 поддерживает интерфейс ERC-20. Мы можем скопировать код интерфейса в проект, а можем использовать существующую библиотеку смарт-контрактов OpenZeppelin. Добавим её в наш проект:

```
npm add @openzeppelin/contracts
```

Теперь сделаем интерфейс токенов ERC-20 доступным для вызова из нашего контракта Poster. Для этого добавьте следующую строку после строки с версией Силидита:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.21 <0.9.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

// ... остальной код контракта
```

По задаче нам необходимо сделать так, чтобы только владельцы заданного токена с заданным балансом могли получить возможность создавать сообщения. Это значит, что нам нужно вызвать функцию `balanceOf` этого токена и проверить, больше ли он нашего порога `threshold` при создании сообщения. Модифицируем соответствующую функцию `post`:

```
function post(string memory content, string memory tag) public {
    IERC20 token = IERC20(tokenAddress);
    uint256 balance = token.balanceOf(msg.sender);
    if (balance < threshold) revert("Not enough tokens");
    emit NewPost(msg.sender, content, tag);
}
```

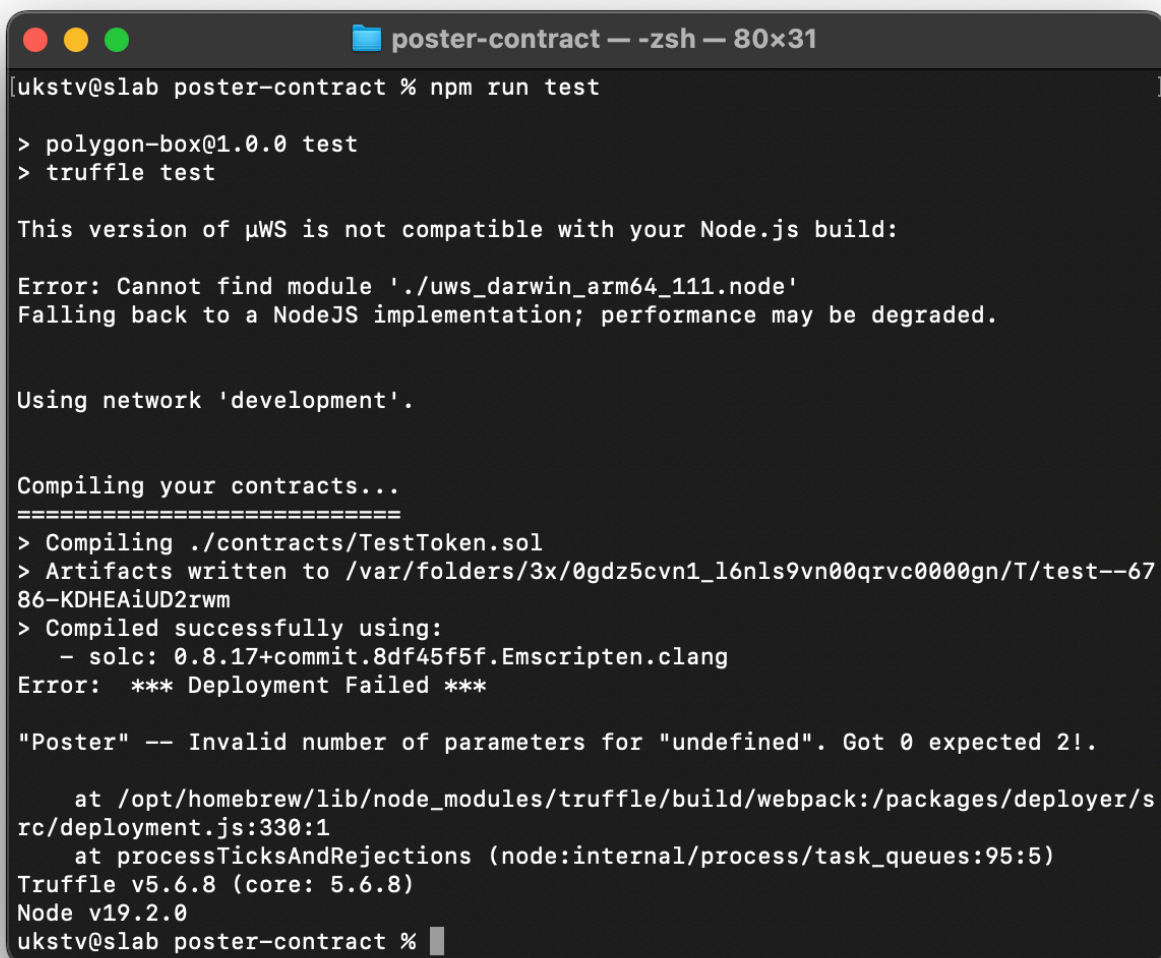
Обратите внимание на строку `IERC20 token = IERC20(tokenAddress);`. Таким образом мы говорим Solidity, что по адресу `tokenAddress` находится контракт, соответствующий интерфейсу ERC-20. Выражение `IERC20(tokenAddress)` обеспечивает корректную кодировку байткода для функций, соответствующим ABI стандарта ERC-20, но из Solidity, то есть внутри блокчейна. В какой-то степени это аналогично вызову функции контракта из-вне блокчейна через ABI.

Мы можем проверить формальную корректность кода, скомпилировав контракты:

```
npm run compile
```

Теперь мы должны протестировать получившийся код. Тестирование производится в изолированной среде "блокчейна" Ganache. Нам необходим контракт токена ERC-20, который мы могли бы использовать в этой среде. Сделаем такой контракт в `contracts/Token.sol`, скопировав его из лабораторной работы 3.

Мы изменили параметры конструктора `Poster` и функцию `post`. Существующий тест должен свалиться с ошибками, поскольку ни тест, ни миграция, доставшаяся от работы 2, не соответствуют измененному коду контракта.



```
poster-contract — -zsh — 80x31
[ukstv@slab poster-contract % npm run test]

> polygon-box@1.0.0 test
> truffle test

This version of uWS is not compatible with your Node.js build:

Error: Cannot find module './uws_darwin_arm64_111.node'
Falling back to a NodeJS implementation; performance may be degraded.

Using network 'development'.

Compiling your contracts...
=====
> Compiling ./contracts/TestToken.sol
> Artifacts written to /var/folders/3x/0gdz5cvn1_l6nls9vn00qrv0000gn/T/test--6786-KDHEAiUD2rwm
> Compiled successfully using:
   - solc: 0.8.17+commit.8df45f5f.Emscripten.clang
Error: *** Deployment Failed ***

"Poster" -- Invalid number of parameters for "undefined". Got 0 expected 2!.

    at /opt/homebrew/lib/node_modules/truffle/build/webpack:/packages/deployer/src/deployment.js:330:1
    at processTicksAndRejections (node:internal/process/task_queues:95:5)
Truffle v5.6.8 (core: 5.6.8)
Node v19.2.0
ukstv@slab poster-contract %
```

Теперь модифицируем миграцию `migrations/2_poster.js`, задав нулевые значения для новых параметров, чтобы миграции не мешали нам запускать тесты:

```

const Poster = artifacts.require("Poster");

module.exports = function (deployer) {
  deployer.deploy(Poster, "0x0000000000000000000000000000000000", 0);
};

```

Модифицируем тест, запустив в него адрес токена и пороговое значение:

```

const Poster = artifacts.require("Poster.sol");
const Token = artifacts.require("Token.sol");

const THRESHOLD = 10;

contract("Poster", (accounts) => {
  it("post ok", async () => {
    const creator = accounts[0];
    const tokenInstance = await Token.new("TestToken", "TTKN", 100, {
      from: creator,
    });
    const posterInstance = await Poster.new(tokenInstance.address, THRESHOLD, {
      from: creator,
    });
    assert.equal(await tokenInstance.balanceOf(creator).then(BigInt), 100n);
    const eventsBefore = await posterInstance.getPastEvents("NewPost");
    assert.deepEqual(eventsBefore, []);
    const content = "Hello, world!";
    const tag = "hello";
    await posterInstance.post(content, tag, { from: creator });
    const eventsAfter = await posterInstance.getPastEvents("NewPost");
    assert.equal(eventsAfter.length, 1);
    const postedEvent = eventsAfter[0];
    assert.equal(postedEvent.args.user, creator);
    assert.equal(postedEvent.args.content, content);
    assert.equal(postedEvent.args.tag, web3.utils.keccak256(tag));
  });

  it("not enough tokens", async () => {
    const creator = accounts[0];
    const malicious = accounts[1];
    const tokenInstance = await Token.new("TestToken", "TTKN", 100, {
      from: creator,
    });
    const posterInstance = await Poster.new(tokenInstance.address, THRESHOLD, {
      from: creator,
    });
    assert.equal(await tokenInstance.balanceOf(creator).then(BigInt), 100n);
    assert.equal(await tokenInstance.balanceOf(malicious).then(BigInt), 0n);
    const content = "Hello, world!";

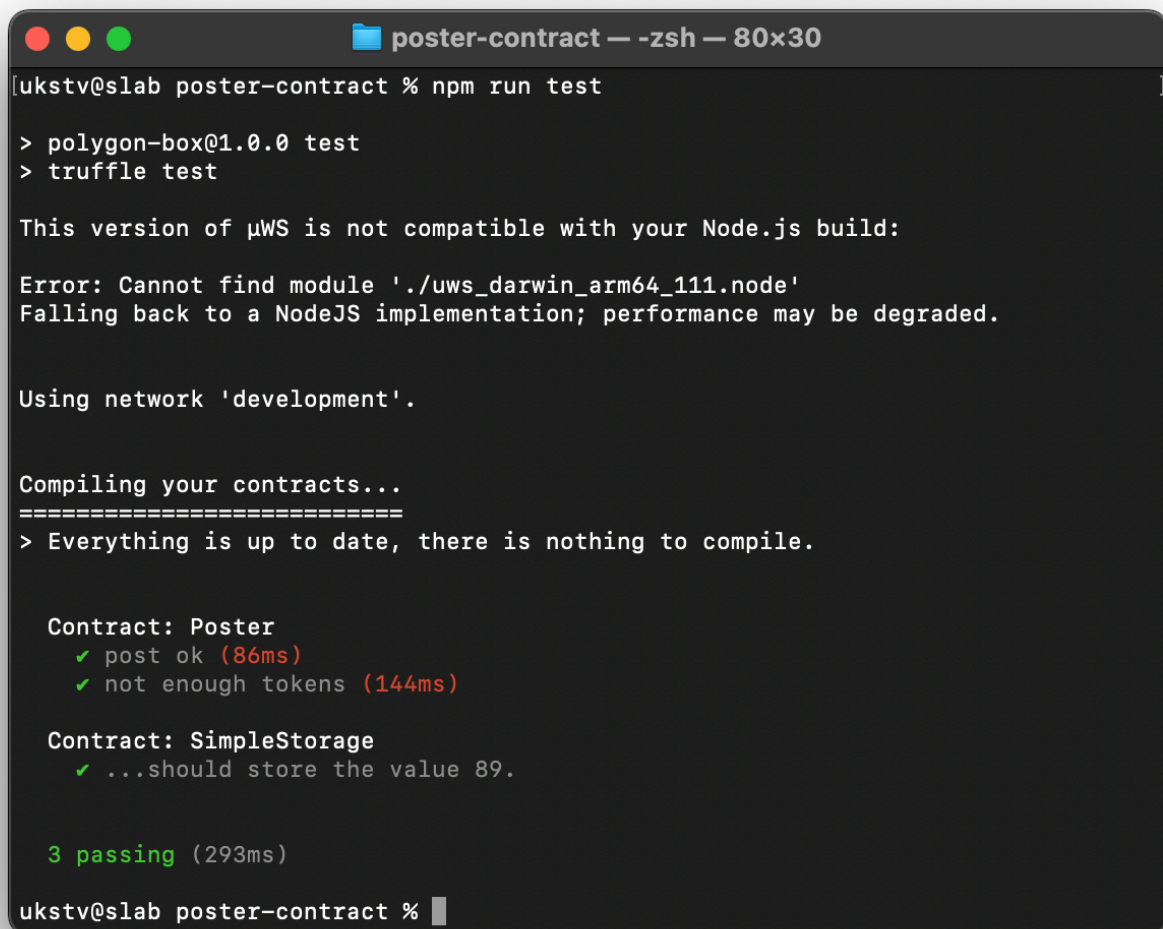
```

```

const tag = "hello";
try {
  await posterInstance.post(content, tag, { from: malicious });
  assert.unreachable('Should throw')
} catch (e) {
  assert.ok(e);
}
});
});

```

Если всё сделано правильно, тест должен сработать успешно.



```

poster-contract — -zsh — 80x30
[ukstv@slab poster-contract % npm run test

> polygon-box@1.0.0 test
> truffle test

This version of μWS is not compatible with your Node.js build:

Error: Cannot find module './uws_darwin_arm64_111.node'
Falling back to a NodeJS implementation; performance may be degraded.

Using network 'development'.

Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.

Contract: Poster
  ✓ post ok (86ms)
  ✓ not enough tokens (144ms)

Contract: SimpleStorage
  ✓ ...should store the value 89.

3 passing (293ms)
ukstv@slab poster-contract %

```

Мы можем развернуть получившийся контракт Poster на блокчейн, однако с текущими нулевыми параметрами никто не сможет запостить текст в контракт. Нам нужно предоставить возможность изменить пороговое значение и используемый адрес контракта. Для этого воспользуемся фрагментом кода из лабораторной работы 3, задающего владельца контракта:

```

// ... предыдущее содержание
address public owner;

```



```

event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

constructor(address _tokenAddress, uint256 _threshold) {
    // ... предыдущее содержание
    owner = _msgSender();
    emit OwnershipTransferred(address(0x0), owner);
}

modifier onlyOwner() {
    require(owner == _msgSender(), "Ownable: caller is not the owner");
    _;
}

function transferOwnership(address _newOwner) public virtual onlyOwner {
    address oldOwner = owner;
    owner = _newOwner;
    emit OwnershipTransferred(oldOwner, _newOwner);
}

// ... предыдущее содержание

```

Задание Добавить функцию `setTokenAddress(address _newTokenAddress)`, меняющую переменную контракта `tokenAddress` на значение `_newTokenAddress`. Вызывать эту функцию может только владелец контракта. Вы можете воспользоваться модификатором `onlyOwner`.

Задание Добавить функцию `setThreshold(uint256 _newThreshold)`, меняющую переменную контракта `threshold` на значение `_newThreshold`. Вызывать эту функцию может только владелец контракта. Вы можете воспользоваться модификатором `onlyOwner`.

Получившийся контракт вы можете развернуть на блокчейне.

2. Развернуть контракт на блокчейне

Воспользуйтесь шагами из лабораторной работы 2 для разворачивания контракта на блокчейне и его верификации на блокчейн-эксплорере.

В рамках развёртывания контракта вам может показаться удобным сменить владельца контракта на ваш адрес в Метамаске, используя функцию `transferOwnership`. Для этого воспользуйтесь следующей миграцией `migrations/2_poster.js`:

```

const Poster = artifacts.require("Poster");

module.exports = async function (deployer) {
    const deployment = deployer.deploy(Poster,
    "0x0000000000000000000000000000000000000000", 0);
    const instance = await deployment.await
    const newOwner = '<ваш-адрес-в-метамаске>'
    await instance.transferOwnership(newOwner)
};

```

После собственно развёртывания контракта вызовите

- функцию `setTokenAddress` с адресом вашего токена, развёрнутого в рамках работы 3;
- функцию `setThreshold` с пороговым значением в 10 токенов.

Обратите внимание, что значение знаков после запятой устанавливается в переменной `decimals` у контракта с токенами (см. стандарт ERC-20). 10 токенов - это $10 * (10^{\text{decimals}})$ единиц баланса.

3. Создать пользовательский интерфейс для взаимодействия со смарт-контрактом

Теперь задача - создать пользовательский интерфейс для работы с контрактом. Вы можете переиспользовать код, разработанный в рамках работы 2. Единственная разница в том, что перед постингом сообщения в контракт ваш код должен проверить соответствие баланса пользователя пороговому значению. Если баланс пользователя больше порогового значения, создавайте транзакцию. Если же баланс меньше, нужно вывести предупреждение пользователю.

Обратите внимание, что проверка баланса должна осуществляться у контракта токена, записанного в переменной `tokenAddress` контракта Poster.

Вопросы для проверки

1. Что означает быть "владельцем" контракта?
2. Могут ли несколько человек быть "владельцами" контракта? Каким образом?
3. Что произойдёт, если контракт из переменной `tokenAddress` не соответствует стандарту ERC20?
4. Что произойдёт, если контракт из переменной `tokenAddress` возвращает некорректный баланс?
5. Как может контракт реализовывать несколько интерфейсов?

Формат предоставления отчёта

В результате работы у вас должно получиться обновленное приложение из работы 2. Новая функция - token-gating, то есть разрешение действий пользователей в зависимости от наличия необходимого баланса токенов. Код приложения - контракты и пользовательский интерфейс - должен быть доступен в системе контроля версий: GitHub, GitLab, Radicle и т.д.

По завершении кодирования попросите своих одноклассников воспользоваться вашим приложением.

Отчёт должен содержать:

- ссылку на верифицированный контракт на блокчейн-эксплорере,
- ссылку на репозиторий,
- описание вашего процесса работы над приложением,
- ответы на вопросы,
- описание приложения как краткое руководство пользователя.