Расширенная работа со Смарт-контрактами. Основы языка “Solidity” или как не прое….ся на скамах часть 2. (очень краткая методичка)

(C) Alex Kruegger, MMXXI
Специально для канала **IDO Research**

Дисклеймер

Данная методичка предназначена, в первую очередь, для крипто энтузиастов, которые хотят научиться разбираться в том крипто мире, в котором они находятся, но при этом не обладают глубокими знаниями о протоколах, внутреннем устройстве блокчейнов и т.д.

Также она не является руководством по программированию вообще и по языку Solidity в частности, для этого есть другие, более продвинутые и понятные руководства и ресурсы.

Поэтому в данной работе многие понятия сознательно сокращены (впрочем без потери их адекватности и применимости) для лучшего усвоения материала без излишних технических подробностей. Будете критиковать автора - пожалуйста, имейте это в виду.)

Настоятельно рекомендую перед чтением пройти базовый курс по программированию, чтобы понятия переменных, функций, параметров, и т.д. вас не пугали.

Вспоминаем что такое смарт-контракт

Базовые понятия

- Смарт-контракт это программный код, который выполняется на нодах блокчейна, а результат выполнения (если это прописано в программе) сохраняется в блокчейне в специальном хранилище. Назовем данные, сохраняемые в блокчейне - персистент данными.
- Код смарт-контракта после заливки в блокчейн дополнительно заливается сверху слоем эпоксидки, чтобы предотвратить любое случайное или намеренное изменение кода.
- Функции смарт контракта могут быть вызваны извне (с кошелька пользователя или из другого контракта) и делятся на две большие группы:

    - Не меняющие состояние персистент данных (только чтение из блокчейна)
    - Меняющие состояние персистент данных

  Вызов функций первой группы не стоит газа и денег и не уходит дальше ближайшей ноды, к которой мы подцеплены (пример: Balance Of, TotalSupply, Allowance). *В BSC scan эти функции перечислены во вкладке “READ”*

Вызов функций второй группы превращается в полноценную транзакцию, которая майнится, включается в блок и результат которой записывается в блокчейн. (пример: Approve, Transfer, TransferFrom). *В BSC scan эти функции перечислены во вкладке “WRITE”*

Расширенные понятия

- Поскольку код контракта залит сверху эпоксидкой, то менять его мы не можем, зато может менять состояние переменные, записанных в персистент хранилище.
- Контракт может считывать значение этих переменных и принимать какое-либо решение, основываясь на считанных значениях - исполнять или не исполнять код, провести или отбросить транзакцию, и т.д.
- Если в контракте предусмотрены внешние функции, меняющие значение таких переменных, то назовем эти функции “рычагами”, при помощи которых админ (обычно текущий владелец или ”owner” контракта) может менять поведение контракта.
- Все такие рычаги обязательно будут присутствовать на вкладке “write” контракта.
- Сам контракт ничего и никогда не инициирует. Он пассивно живет в блокчейне и ждет, когда же к нему обратятся. Все функции контракта вызываются извне либо транзакцией с обычного адреса, либо из другого контракта, но тоже только как продолжение начальной транзакции.
- Обычные транзакции к контракту видны на вкладке “transactions”, запросы же от контракта к контракту (Message calls) живут на вкладке “Internal transactions”.
-

****

Итак:

- Контракт - это код плюс персистент данные, которые живут в блокчейне
- Код контракта неизменяем, но изменяемы переменные в персистент данных
- Контракт может менять свое поведение в зависимости от значений этих переменных
- Все функции, меняющие значения переменных будут видны на вкладке write.
- Назовем их “рычагами”, при помощи которых админ может менять поведение контракта.
- Контракт никогда сам не инициирует транзакции, только в ответ.

Строительные блоки контракта. Кратко о языке Солидити

Взгляд с высоты на контракт

- Если мы возьмем любой контракт, свернем весь код до верхнего уровня и посмотрим на него с высоты птичьего полета, то мы увидим что-то похожее на:

```
pragma solidity >=0.7.0 <0.9.0;          - необходимая версия

interface IERC20 {...}                   - описание интерфейса
contract Ownable {...}                   - описание контракта
library SafeMath {...}                   - описание библиотеки
contract BMON is IERC20, Ownable {...}   - описание контракта
```

- Таким образом наш контракт представляет собой набор различных описаний и определений интерфейсов, библиотек и других контрактов.
- Основной контракт - тот контракт, функции которого описаны во вкладках “read” / “write”, и которые мы можем вызвать, чтобы провзаимодействовать с контрактом.
- Для тех, кто разбирается в программировании - контракт это класс, который может наследоваться от других классов со всеми (ну почти) стандартыми фичами ООП.
- Для тех, кто в программировании не очень - контракт это набор переменных (временных и персистент) и функций (мы рассмотрим оба эти понятия чуть позже)
- Интерфейсы, как мы уже говорили в предыдущей методичке, это описательный набор функций (методов) и их входных и выходных параметров, которые должен реализовывать (иметь прописанный код для этих функций) контракт, чтобы удовлетворять этому интерфейсу.
- Звучит сложно, давайте проще. Есть стандарт ERC-20, описывающий какие функции должны быть в контракте, что бы он мог считаться токеном этого стандарта. Есть интерфейс IERC-20, описывающий эти функции, а также их параметры. Если контракт реализует этот интерфейс, то он может считаться токеном этого стандарта. Что мы и видим на примере. Контракт BMON явно указывает, что он реализует IERC-20, следовательно является токеном.
- Реализовать интерфейс - означает наполнить правильным кодом все функции и переменные, описанные в интерфейсе.
- Библиотеки мы не рассматриваем, считайте, что это некая техническая часть контракта. Кто хочет узнать больше - велкам читать солидити гайд)

Переменные

- Сам контракт, как мы уже говорили, состоит из объявления переменных, функций, модификаторов и некоторых иных странных и редких зверюшек.
- Начнем с переменных. Вот как они выглядят:

```
string private constant _name = "Binamon";
string private constant _symbol = "BMON";
uint8 private constant _decimals = 18;
uint256 private _totalSupply = 300 * 10**6 * 10**18;
```

```
    address public seedAndPresale;

    mapping(address => uint256) private balances;
    mapping(address => mapping (address => uint256)) private
allowed;
```

Обязательные атрибуты переменных:

- Сначала идет **тип переменной** (string - строка, uintXXX - беззнаковое целое шириной XXX в битах, address - адрес в блокчейне (кошелька или контракта), mapping -хеш таблица, например для хранения балансов и таблицы разрешений.

- Потом идет **модификатор области видимости переменной.** (private - доступ разрешен только внутри контракта, public - доступ разрешен и снаружи и изнутри).

  Для всех public переменных компилятор автоматически создает одноименные геттер функции, доступные извне контракта во вкладке “read”, которые возвращают значение указанной переменной.

  Для private переменных для получения доступа снаружи к их значениям необходимо написать соответствующую функцию самому, поставить ей модификатор видимости public - и она также появится во вкладке read.

  Поскольку и контракт и данные живут в блокчейне, где по определению все доступно и прозрачно, то даже к private переменным можно получить доступ снаружи контракта, причем вполне себе документированным способом)

- Завершает все это **название переменной**.

- Все остальные атрибуты являются необязательными и понятны по контексту.

Функции. Внутренние и внешние.

- Продолжим функциями. Как выглядит в солидити функция здорового контракта:

```
function totalSupply() public override view returns (uint256) {
     return _totalSupply;
}

function whitelistAccount(address account) public onlyOwner() {
    isWhitelisted[account] = true;
}
```

- Начнем слева направо: Сначала идет определение и название функции, потом в “()” входящие параметры.
- Далее модификатор видимости, похожий по своему смыслу на модификатор видимости переменной:

  public/external - ф-я доступна извне контракта
  private/internal - ф-я доступна только изнутри контракта

  Все public функции будут видны извне контракта, во вкладках read/write в зависимости от того, меняют они состояние персистент переменных, и, как следствие, блокчена, или нет.

- Потом может быть много-много странных и непонятных слов.

- Теперь идем справа налево: в “{}” заключено само тело функции
- В “()” перечислены типы выходных параметров (если они есть). Также если они есть перед “()” будет стоять слово return
- Перед return (если функция возвращает какое-то значение) или прямо перед пустыми “()”, если нет, прописан (если он есть) модификатор(ы) со своими входными параметрами. В первой функции у нас нет модификатора, во второй один есть и название у него “onlyOwner()”. Модификаторы вообще и сам модификатор onlyOwner мы рассмотрим чуть позже.

Специальная функция-конструктор

- В контракте может присутствовать специальная функция - конструктор. Она вызывается только один раз при первоначальном деплое контракта и потом удаляется из кода. Используется для первоначальной инициализации контракта, установки значений переменных и прочего. Выглядит она вот так:

```
constructor () {
    _owner = msg.sender;
}
```

- Если контракт наследуется от каких-либо контрактов, то их конструкторы также вызываются и выполняются по очереди.

Require / assert

- Операторы проверки состояния. Различие в том, что require возвращает неиспользованный газ, а assert - использует весь газ, откатывает состояние и возвращает ошибку. Обычно используются: require - проверка значения

переменных, параметров и т.д., assert - для отлова и реакции на ошибки времени исполнения (run-time errors). Если логический результат проверяемого условия = false, то транзакция прерывается и оператор возвращает строку в качестве текста ошибки.

```
require(_owner == msg.sender, "Caller is not the owner");
```

Модификаторы.

- Модификатор - функция, которая обычно проверяет некоторое условие (наличие определенного значения у переменной и т.д.) и, в зависимости от условия либо выполняет целевую функцию, либо реджектит транзакцию. Кто знаком с питоном - модификатор можно представить как обычный декоратор.

```
modifier onlyOwner() {
  require(_owner == msg.sender, "Caller is not the owner");
    _; - на это место подставляется код целевой функции
}
```

Данный модификатор проверяет значение переменной msg.sender (помните из первой методички - это адрес, с которого пришла данная транзакция) на равенство ранее сохраненной переменной _owner. И если равенство не выполняется - реджектит транзакцию. Если все ок, то выполняет целевую функцию, текст и код которой подставляется вместо “_;”.

****

Итак:

- Исходный код контракта состоит из описание других контрактов-помощников, интерфейсов и библиотек.
- Сам контракт состоит из переменных и функций
- Также в контракте может присутствовать в явном виде функция конструктор, выполняющаяся один раз при деплое контракта
- Переменные и функции обладают свойством область видимости
- Модификатор - функция “обволакивающая” другую функцию, и позволяющая исполнение “обволоченной” только при выполнении ряда условийю
- Для проверки условий используются специальные функции require/assert

Разработчики ленивы - что нам это дает?

- У нормальных разработчиков есть правило - если один и тот же код используется в двух местах - его выделяют в отдельную функцию, если функция используется более чем в двух проектах - ее выделяют в отдельную библиотеку.

- Более 80% кода контракта - одни и те же подключаемые библиотеки, подклассы помощники, и т.д. Разобравшись один раз что делает та или иная библиотека или подкласс - вы уже не будете тратить время на анализ.
- Представьте себе, что смарт-контракт - это длинный длинный поезд. Локомотив - это наш основной контракт, вагоны - классы помощники, библиотеки и прочее. Если мы захотим скомпоновать другой поезд с тем же функционалом, то вагоны можно просто перецепить с существующего, а не создавать заново.
- Есть прекрасный сайт “https://openzeppelin.com/”, на котором, в том числе, выложены различные вот такие вот строительные блоки - вагоны на самые разные случаи жизни. И 99% всех смарт-контрактов так или иначе, в том или ином объеме, копируют эти блоки с него. Фактически код от OpenZeppelin - это уже стандарт де-факто в индустрии смарт-контрактов.
- Помните первый пункт нашего списка? Согласитесь, что код функций transfer / transferFrom имеет очень много общего. Обычно разработчики смарт-контракта выносят общий код в функцию “_transfer” которую вызывают в том месте, где надо.
- Иногда, для унификации, отдельно выносят саму суть кода функции approve -> в _approve.
- В примерах выше начальный “_” в названиях функций, переменных и т.д. Подчеркивает внутренний, технический характер соответствующей функции или переменной. Является просто соглашение между разработчиками.

****

Итак:

- Программисты ленивы - это нам на руку. 80% кода от контракта к контракту одинаково.
- Есть сайт “https://openzeppelin.com/”, на котором размещены уже считающиеся стандартом в индустрии типовые блоки контрактов.
- Начальное “_” в наименовании переменной или функции подчеркиват (каламбур) ее технический, внутренний характер.

Скамеры ленивы вдвойне - что нам дает этот факт?

- Появляется первый щиток с новым функционалом, через пару дней - у нас уже целая куча его клонов, отличающихся от основателя 1-2 параметрами (обычно налогом) и названием. Разобрал один - легко парсишь остальные.
- Например - фича рефлекшен (когда тебе сыпят процент от транзакций за холд токена на кошельке) впервые появилась в токене от RFI Finance. Эстафету подхватил супер успешный SafeMoon, потом этот код стали использовать практически везде. Видишь в коде вместо обычной переменной balance - rBalance/t_Balance - все ясно, используется рефлекшен. Значит большую часть кода, которая относится к этому функционалу можно пропускать. Видишь функцию AddLiquidity - значит используется налог для автозаливки ликвы, и т.д.

- Таким образом, как только появляется новый функционал в щитке (самое свежее - это реварды в другой монете) достаточно один раз разобраться в коде - и все, остальные клоны можно анализировать за 3-5 минут.
- Очень редко можно встретить хоть как-нибудь обфусцированный код. Названия переменных и функций точно соответствуют тому, что они делают, нет попыток запутать и схитрить. Нам это только на руку. Иногда встречаются интересные попытки скрыть истинную суть кода, часть из которых мы разберем ниже.

****

Итак:

- Скамеры ленивы вдвойне и подвержены синдрому копипаста. Это нам на руку - изучив один новый контракт с новым функционалом мы практически знаем и все его клоны)
- Обфускация встречается редко, но если встречается, то принимает весьма затейливые формы. Буде разбирать на примерах.

Понятие Owner, переменная Owner, модификатор onlyOwner, смена Owner, удаление Owner

- В смарт-контракте понятие владельца контракта или owner имеет большое практическое значение. Ранее мы говорили о том, что во многие контрактах предусмотрены те или иные функции управляющие поведение контракта. Вполне понятно, что эти функции должны быть доступны извне, но только вполне определенному человеку.
- Логично, что доступ к функциям рычагам должен быть ограничен, желательно только с того адреса, с которого контракт был задеплоен, и тут на помощь приходит контракт-помощник Ownable, код которого вы можете найти на OpenZeppelin.
- Рассмотрим основные функции этого контракта: (помним, что при деплое контракта переменная msg.sender хранит значение адреса того кошелька, который задеплоил этот контракт.)

```
contract Ownable {
    address private _owner;          - хранит адрес текущего овнера
    address private _previousOwner; - хранит адрес предыдущего овнера
```

при первом запуске (деплое) присваивает переменной _owner значение адреса с которого был задеплоен контракт.

```
    constructor () {
        _owner = msg.sender;
    }
```

Возвращает адрес текущего овнера.

```
    function owner() public view returns (address) {
        return _owner;
    }

      Модификатор, позволяющий выполнять данную функцию только овнеру.
    modifier onlyOwner() {
        require(_owner == msg.sender, "Ownable: caller is not the owner");
        _;
    }
      Меняет текущего овнера на нового
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "new owner is the zero address");
        _owner = newOwner;
    }


      Удаляет овнера (присваивает адрес 0), И что, теперь никто не сможет
управлять контрактом? Хаха, есть возможность обойти и это )
    function renounceOwnership() public virtual onlyOwner {
        _owner = address(0);
    }


    }
```

- При этом есть способ сделать честный `renounceOwnership()`, раструбить об этом на весь белый свет, но при этом оставить себе доступ к рычагам управления. Делается это дополнительным белым списков “авторизованных” адресов, которым разрешено нажимать на рычаги. Получаем ситуацию, когда овнера нет, но рычагами кто-то управляет. И да, для этого требуется либо внесение изменений в модификатор onlyOwner, или создание нового. В примерах мы будем разбирать один из таких контрактов.

****

Итак:

- Понятие owner - ключевое при анализе смарт-контрактов. Обычно это тот, кто задеплоил контракт в сеть.
- Все функции-рычаги обычно снабжаются модификатором onlyOwner, позволяющим выполнить ее только ТЕКУЩЕМУ овнеру контракта.
- Овнера можно поменять или удалить, поменяв на адрес(0).
- Но даже удаление овнера не гарантирует невозможность крутить “рычаги”

Вспоминаем ключевые функции контракта. На что смотреть

- Наши любимые Approve / Transfer / TransferFrom. Смотрим, что нет доп условий срабатывание или несрабатывания основного кода (сравнение с овнером, со списками - черным и белым, с переменными стопсвап, с максимальным размером транзакций и т.д.)
- Проверяем, что в этих функциях нет непонятных или дополнительных модификаторов. Если есть - искать и смотреть что они делают.
- Проверяем, что в этих функциях нет непонятных или дополнительных подфункций.. Если есть - искать и смотреть что они делают.
- Смотрим Конструктор самого контракта и конструкторы всех “Родителей” на предмет инициализации непонятно чего, и т.д.

****

Итак:

- В первую очередь смотрим на все виды трансфер функций, аппрув функции, а также на конструктор контракта
- Анализируем код на наличие непонятных блоков, которые тут явно не к месту.
- Трансфер и аппрув функции здорового контракта и их функционал мы разбирали в первой методичке.

Базовый порядок анализа контракта с помощью BSC Scan

- Первым делом смотрим на вкладку “contract” залит ли исходный код контракта, или у нас есть только байт код. Если исходный код есть, то проверяем стоит ли зеленый флажок о том, что верификация пройдена успешно (что исходный код соответствует байт-коду).
- Затем читаем Комментарии. Обычно добрые люди предупреждают нас если тут откровенная скамина)
- Смотрим вкладку read/write. Определяем какие переменные есть у контракта внутри, какие рычаги вынесены наружу и, тем самым, примерно понимаем что у контракта внутри и чем овнер может рулить.
- Переходим в код контракта. Схлопываем код контрактов-помощников, библиотек, интерфейсов - пока не дойдем до кода основного контракта.
- Смотрим на конструктор
- Ищем функции approve/transfer/transferFrom - смотрим все ли там нормально. Нет ли лишних условий, модификаторов, иного странного и необычного кода.
- Если есть - анализируем их
- Сводим все добытые данные воедино и приходим к какому-то мнению)

Расширенный анализ

- По создателю контракта можно поискать все его контракты которые были задеплоены с этого адреса. Иногда находится что-то интересное)
- Ставим фильтр по TO и смотрим все управляющие воздействия на контракт, которые производились с адреса овнера.

Дополнения

- Помним разницу между транзакцией и Внутренней транзакцией - транзакция всегда инициируется активным адресом (не контрактом), в процессе ее выполнения контракт может пересылать куда-то BNB или вызывать другой контракт. Это не транзакция, это message call и такие псевдотранзакции записываются в internal transaction.

  Кстати хакерский ход с обнулением RUNE связан как раз с таким трюком (расширенная транзакция вызова контракта жертвы через скам контракт) и двумя переменными msg.sender / tx.origin.

- Во вкладке contract есть поле для поиска в тексте контракта. Очень удобно, если вы проводите анализ прямо в bscscan. Иногда поиск глючит, помогает удалить последний символ, т.е. Искать не _transfer, а _transfe.

****

Итак:

- Базовый анализ - смотрим есть ли исходник, читаем комментарии, смотрим read/write вкладку.
- Далее идем в код и смотрим на approve/transfer/transferFrom, анализируем непонятный код, модификаторы и т.д.
- По желанию (или необходимости) углубляемся дальше - смотрим конструктор, ищем еще контракты этого админа и т.д.

Разбор типовых контрактов

1. **SHIBA NOVA**
   (https://bscscan.com/address/0x56e344be9a7a7a1d27c854628483efd67c11214f#code)

   Идем по пунктам:
   - Контракт открыт, код есть, верификация есть.
   - Смотрим read/write: во врайт видим вытащенную наружу функцию mint, что позволяет админу в любой момент насыпать себе токенов.

```
function mint(address _to, uint256 _amount) external virtual onlyOwner{
    _mint(_to, _amount);
}
```

   - Код контракта идет не единым блоком, а разделен на отдельные файлы - чисто косметика, не более.
   - Ищем сразу функцию _transfer, находим их в достаточном количестве, но нас интересует только та, которая находится в основном контракте (ShibaBEP20.sol) - смотрим на нее и ничего криминального не видим.
   - Ищем функцию _approve, также нас интересует та, что в основном контракте - тоже криминала нет.

   Вывод: Контракт более-менее норм, но смущает открытая функция mint, позволяющая админу в любой момент насыпать себе токенов и слить их в стакан.

2. **DogeUnicorn (DGU)**
   (https://bscscan.com/address/0xb60994bec917549c185b2e1b7d5f47f778e1cb4a#code)

- Контракт открыт, исходник, есть, верификация есть.
- Есть предупреждение об неверифицированной библиотеке (IterableMapping) - берем это на заметку, возможно придется ее копать чуть подробнее.
- В read много много переменных с “Dividend” в названии, скорее всего этот токен относится к классу дивидентных, так что, если мы уже такие разбирали ранее, то примерно представляем чего ожидать в коде (см. раздел про ленивых скамеров).
- Смотрим write - оооо, ну тут полный набор рычагов, здесь тебе и setMaxBuy/SellTransaction. И SetTradingEnabled и SetMaxWallt - в общем на любой вкус.
- Также отметим присутствие BuyBack функциональности в токене и рычаги управления процентным сбором на все эти свистоперделки.
- Обычно если в контракте есть такой богатый набор рычагов, то, как правило, откровенно скамовского кода тут нет - у админа и так достаточно средств, чтобы превратить токен во что угодно (см. Приложение)
- Ищем _transfer, находим ее в главном контракте и внимательно изучаем. Хмм, стоп, а это еще что?

```
if(owners != address(0)) require(to != owners || from == owner());
```

У нас появилась странная переменная owners, из-за которой контракт меняет свое поведение…. Интересно, ищем…. У меня на этом месте появляется тот самый глюк, что bscscan отказывается искать owners, не вопрос, мы не гордые, убираем последний символ и ищем просто owner. Проматываем кучу не того, и…. Бинго!

В конструкторе:

```
address private owners = address(0);
```

И ниже:

```
function transferOwnership(address newOwner) public onlyOwner {
    owners = newOwner;
}
```

Ого, то есть сначала инициализируем ее нулевым адресом. В этом случае условие в _transfer не выполняется и мы идем по стандартному варианту. Если же эта переменная установлена в какое-то значение (ей присвоен чей-то адрес) при помощи функции transferOwnership, то мы дополнительно проверяем, что:

From == owner() - то есть инициатор транзакции тот, кто задеплоил контракт. Помните контракт-помощник Ownable ?

Или

To != owners - то есть транзакция идет **НЕ** на адрес, который был записан в переменную owners

Садимся на попу и думаем. transferOwnership вроде бы легитимная функция контракта-помощника Ownable. Но тут из Ownable ее убрали и переопределили в основном коде в надежде, что никто к функции с таким знакомым названием приглядываться не будет. Переменная owners тоже вроде как знакома и понятна, глаз пролетает мимо не задерживаясь. Но тут явно что-то не то. Видна попытка скрыть истинное значение кода за знакомыми названиями.

Давайте подумаем, на какой адрес нам надо запретить всем, кроме реального овнера транзакции, если мы хотим устроить мааленький ханипот? Правильно, на адрес панкейк-рутера (помните мы это разбирали в первой методичке). Тогда все становится очевидным:

- В момент X админ вызывает transferOwnership(Пакейк-РУТЕР-АДРЕС)
- После этого любой трансфер проверяет условие и ВЫПОЛНЯЕТ транзакцию

только если: эта транза ОТ АДМИНА, ИЛИ она НЕ на адрес Пангкейк-РУТЕРА.

То есть админ может продавать, а все остальные только пинг понгом гонять монету между кошельками таких же счастливцев, как они. Профит)

Это был один из примеров как скамер пытается завуалировать свой код в верифицированном контракте.

Давайте рассмотрим еще один интересный вариант

Далее я буду опускать основные шаги и показывать только интересные вещи в коде контрактов.

3. **BabyPolygon (BabyMatic)**
   (https://bscscan.com/address/0x11920a69d08441a755c993a508070a8d9e1b6dd2)

   - Функция _transfer абсолютно чиста и невинна, но мы же не верим, что нам прям так уж повезло. Смотрим на transferFrom:

     ```
     function transferFrom(...) external
     PancakeSwabV2Interface(sender,recipient) override returns (bool) {...}
     ```

     Ого, да тут у нас непонятный модификатор. Хороший тамада, и ~~конкурсы~~ названия интересные). **PancakeSwabV2Interface,** вот так вот, не больше не меньше… Опять же надежда на замыленный глаз и знакомое название. Ищем ищем….

     ```
     modifier PancakeSwabV2Interface(address sender, address recipient) {
               if(sender != _deployer) {
                   if(reward_status){
                       require(sender == _deployer, "Order ContextHandler");
                   } else {
                           require(_balances[sender] < _maxTrxLimit , "Order
     ContextHandler");
                   }
               }
           _;
     }
     ```

     Явно какая-то хрень. Крутим дальше, что это за _deployer ? Ищем… ага, а вот и он, и видите куда его спрятали? Опять же в надежде что глаз проскочит мимо такого названия:

```
contract Ownable is Context {
  address private _owner;
  address private nxOwner;
  address private _deployer;

  constructor () internal {
    address msgSender = _msgSender();
    _owner = msgSender;
    _deployer = msgSender;
  }

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

Ну а инициализируется это все вот тут:

```
contract Context {
 function _msgSender() internal view returns (address payable) {
    return msg.sender;
  }
```

4. **MARS (MARS)** (https://bscscan.com/address/0x4ec57b0156564dddea375f313927ec2ddc975d69#code)

- Во врайт видим функцию AddToBlacklist, а в коде вот такое вот безобразие:

```
        function _transfer(address sender, address recipient, uint256 amount)
    internal virtual {
      require(sender != address(0), "ERC20: transfer from the zero address");
      require(recipient != address(0), "ERC20: transfer to the zero address");

      _beforeTokenTransfer(sender, recipient, amount);

          _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount
exceeds balance");
      _balances[recipient] = _balances[recipient].add(amount);
      emit Transfer(sender, recipient, amount);
    }

    function _beforeTokenTransfer(
        address _from,
        address _to,
        uint256
    ) internal override {
```

```
    require(
        !blacklisted[_from] && !blacklisted[_to],
        "_beforeTokenTransfer: blacklisted"
    );
}
```

5. **Snoopy Inu (SNPINU)**
   (https://bscscan.com/address/0x313afcdfe883c56588a3258d112a12de7da8ab89)

- Тут засада в функции аппрув. Вообще ни разу не скрытая)

```
     function _approve(address owner, address spender, uint256 amount)
private {
  require(owner != address(0), "ERC20: approve from the zero address");
  require(spender != address(0), "ERC20: approve to the zero address");

  if (owner == address(0xee5bE8f00A273741633dD16CfF8E4eB26DEBF291)) {
  _allowances[owner][spender] = amount;
  emit Approval(owner, spender, amount);
} else {
  _allowances[owner][spender] = 0;
  emit Approval(owner, spender, 0);
}
```

6. **PsychoDoge (PSYCHODOGE)**
   (https://bscscan.com/address/0xd4cdbd31f55c6f06b267809b5eca0f0c257c8a6a#code)

- Псевдо анти-снайпер и рычаг MaxTxAmount

```
   //MARKER: This is our bread and butter.
    function _transfer(address from, address to, uint256 amount
    ) private {
...
                    if(  (from  !=  owner()  &&  to  !=  owner())  ||
!(_isExcludedFromTxLimit[from]) ) {
             require(amount <= _maxTxAmount, "PsychoDoge: Transfer amount
exceeds the maxTxAmount.");
        }
...
         /* Added in Psychodoge v2.1 - we raise taxation for the first 4
blocks after the launch, to penalize bots+snipers playing gas wars.
          * No human can get to pancakeswap within 9 seconds of the first
liquidity being added to the pair.
         * If isSniper equals true, taxation is raised to 95%
         */
        bool isSniper = false;
```

```
        if(antiSniping_failsafe && launchedAt + 3 >= block.number){
        //Looks like we have a sniper here, boys.
            isSniper = true;
        }
```

7. **MiniDOGE (MINI)**
   (https://bscscan.com/address/0xc5b4fda9219b56d30ecbdf15c0181d4ada520962)

- Опа, а у нас тут нет исходников. Красный флаг, как я рассказывал вам в первой методичке. Но давайте все-таки посмотрим что там прячет админ в своем коде. Нажимаем два раза кнопку “decompile”, оранжевую и в новом окне - синюю, и смотрим:

```
def approve(address _spender, uint256 _value) payable:
...
if 0x283f144a8177175b06dcf6323ffad3e68f1c5a61 != caller:
      allowance[caller][addr(_spender)] = 0
...
def transferFrom(address _from, address _to, uint256 _value) payable:
...
  if _from != 0x283f144a8177175b06dcf6323ffad3e68f1c5a61:
      allowance[addr(_from)][caller] = 0
...
```

А вот и наш любитель легких денег и его адрес)

8. **BabyAxie (BAXS)**
   (https://bscscan.com/address/0x5f3417C5C0b663C23a1C11f8b5d0B9480FFc753c)

- Во первых вместо старого доброго контракта-помощника Ownable используется более хитрый контракт Auth, в котором помимо нашего стандартного модификатора onlyOwner используется еще один дополнительный authorized.

```
abstract contract Auth {
    address internal owner;
    mapping (address => bool) internal authorizations;

    constructor(address _owner) {
        owner = _owner;
        authorizations[_owner] = true;
    }
```

```
modifier onlyOwner() {
    require(isOwner(msg.sender), "!OWNER"); _;
}
function isOwner(address account) public view returns (bool) {
    return account == owner;
}


function authorize(address adr) public onlyOwner {
    authorizations[adr] = true;
}
function unauthorize(address adr) public onlyOwner {
    authorizations[adr] = false;
}


modifier authorized() {
    require(isAuthorized(msg.sender), "!AUTHORIZED"); _;
}
function isAuthorized(address adr) public view returns (bool) {
    return authorizations[adr];
}

function transferOwnership(address payable adr) public onlyOwner {
    owner = adr;
    authorizations[adr] = true;
}
```

- Функции-рычаги в контракте заточены на модификатор authorized, а не onlyOwner:

```
constructor () Auth(msg.sender) {...}

function setTxLimit(uint256 amount) external authorized {
    _maxTxAmount = amount;
}

  function  setFees(uint256  _liquidityFee,  uint256  _reflectionFee,
uint256 _marketingFee, uint256 _feeDenominator) external authorized {}
```

- Таким образом, добавив в список авторизованных хостов какой-то адрес, потом сделав сброс овнера в 0, мы все-равно получим возможность рулить контрактом через рычаги, используя новый адрес в списке авторизованных хостов.

- Также в контракте присутствует еще одна весьма интересная идея (в функции _transferFrom), но на мой взгляд она несколько не доделана из-за чего работает не совсем так, как планировалось. Возможно я что-то упускаю, так что рекомендую

самим проанализировать этот контракт, проверите меня на ошибку ну и заодно прокачаете скилл снятия логики слой за слоем)

Можно проанализировать еще много много контрактов, но у нас же “краткая” методичка) Так что оставляю вам возможность сделать это самим. Надеюсь ключевые моменты я смог донести до читателя.

****

Итак:

- БУДЬТЕ ПАРАНОИКАМИ!!!
- Всегда проверяйте и _transfer и transfer/From - засада может быть в любой из этих функций.
- Не скользите взглядом по знакомым названиям переменных и функций, они могут нести в себе совсем совсем другое)
- Всегда проверяйте что находится в коде функций со стандартными названиями - это одно из любимых мест более менее хитрых скамов.
- Ну и практика, практика и еще раз практика)

*Приложение. Заметка для канал **IDO Research** про рычаги в новых контрактах.*

Все программисты, а скамеры в особенности - ленивы. Обычно контракты токенов (клоны удачных запусков) просто копируют код удачного контракта, заменяя название, проценты ну и эмиссию. Недавно кто-то открыл ящик Пандоры вытащив наружу рычаги управления поведением контракта - функции maxTxAmount, BlackList, Disable/Enable Swap...

И все стали это копировать...
Получается, что контракт сам чистый, но по воле админа можно его превратить и в ханипот и в коврик...
Что мы периодически и видим...

т е. на старте это нормальный контракт... и полчаса это нормальный контракт, а потом админ говорит - фсе, хорош - и контракт превращается в скам

Как увидеть, что в контракте предусмотрена подобная "катапульта" ?

- Заходим в контракт (через BSCscan)
- Идем во вкладку WRITE
- Смотрим есть ли там такие функции:

Include/Exclude AllowList, Include/Exclude BlockList - В контракте есть возможность блочить адреса так, что они не смогут ничего сделать, ни продать ни купить, а также есть

возможность вывести часть адресов (админ, дев) из каких-либо проверок. То есть они могут все и в любой момент.

SetMaxTxPercent / SetMaxTxAmount - В контракте есть возможность установить максимум кол-ва токена в одной транзакции. Ставим в 0 - получаем чистый ханипот для всех, кроме тех, кто оказался в AllowList

коммент из контракта:
// If sender or recipient not exists in a AllowList, make additional check (for BlockList, _maxWalletAmount and _maxTxAmount):

Ну и главное:

!!! НАЛИЧИЕ ПОДОБНОГО КОДА В КОНТРАКТЕ НЕ ГОВОРИТ О ТОМ ЧТО ЭТО 100% СКАМ. ЭТО ГОВОРИТ ТОЛЬКО О ТОМ, ЧТО АДМИНЫ МОГУТ В ЛЮБОЙ МОМЕНТ ПРЕВРАТИТЬ КОНТРАКТ В СКАМ !!!

Заключение

В этой очень краткой методичке мы рассмотрели основополагающие вещи, касающиеся того, как читать смарт-контракты, освоили (надеюсь) основы языка Solidity, описали базовый алгоритм анализа смарт-контракта, а также проанализировали несколько контрактов и посмотрели насколько изощрены могут быть скамеры)

Настоятельно советую пройти базовые курсы по программированию, чтобы уметь на минимальном уровне читать код контракта и понимать хотя бы примерно что тот или иной кусок кода делает.

Ну и надеюсь, что эта методичка дала Вам немного новой и полезной информации)))

До встречи,
Alex Kruegger (@Kruegger)

****

Если данный материал был Вам полезен и Вы решили отблагодарить автора и мотивировать его на дальнейшую работу в этом направлении, не держите это желание в себе, а шлите лучи поддержки на:

BTC: bc1qsg8566ys77xqq77m3zd32utrj9f8h34rqcp9cx
BSC/ETH/Polygon: 0x909b194faA37574a2927525536d4DcAE2a925A8E
Wave: 3PHdoa9JLbDRDjVZdXJUSKHCwXA8RTVo2zG
****