Титульный лист

Содержание

Оглавление

[Титульный лист 1](#_Toc8499669)

[Содержание 2](#_Toc8499670)

[Введение 4](#_Toc8499671)

[Основная часть 5](#_Toc8499672)

[1. Общие сведения о безопасности в блокчейн сети 5](#_Toc8499673)

[1.1. Что такое блокчейн? 5](#_Toc8499674)

[1.2. Основы безопасности в блокчейне. 6](#_Toc8499675)

[1.2.1. Введение в криптографию 6](#_Toc8499676)

[1.2.2. Криптография на эллиптических кривых (ECC) 8](#_Toc8499677)

[1.2.3. Уязвимости блокчейна 16](#_Toc8499678)

[1.2.3.1. Уязвимости на уровне сети 16](#_Toc8499679)

[1.2.3.2. Уязвимости на уровне пользователя 16](#_Toc8499680)

[1.2.3.3. Уязвимости на уровне майнинга 16](#_Toc8499681)

[1.2.3.4. Уязвимости, не зависящие от блокчейна 16](#_Toc8499682)

[2. Уязвимости смарт-контрактов 16](#_Toc8499683)

[2.1. Уязвимости в коде контракта 17](#_Toc8499684)

[2.1.1. Re-entrancy 17](#_Toc8499685)

[2.1.2. Access control 19](#_Toc8499686)

[2.1.3. Unchecked return values for low level calls 20](#_Toc8499687)

[2.1.4. Denial of service 22](#_Toc8499688)

[2.1.5. Bad randomness 23](#_Toc8499689)

[2.1.6. Time Manipilation 24](#_Toc8499690)

[2.1.7. Short Address Attack 25](#_Toc8499691)

[2.2. Проблемы в процессе разработки 25](#_Toc8499692)

[2.3. Уязвимости в языке 26](#_Toc8499693)

[2.3.1. Неправильное использование DelegateCall 26](#_Toc8499694)

[2.3.2. Arithmetic Issues 26](#_Toc8499695)

[2.4. Проблемы в концепции 28](#_Toc8499696)

Введение

Основная часть

1. Общие сведения о безопасности в блокчейн сети
   1. Что такое блокчейн?

Блокчейн по своей сути – это логически связанная последовательность информационных блоков (block – блок, chain - цепочка). Каждый блок содержит в себе данные о группе транзакций и заголовок, состоящий из собственного хеша, хеша транзакций и хеша предыдущего блока. Это в первую очередь децентрализованная система, существующая благодаря множеству объединенных в одну одноранговую сеть компьютеров. Блокчейн – это инструмент децентрализованного хранилища данных, обеспечивающий высокий уровень безопасности.

Такая система организации данных также обеспечивает прозрачность транзакций и каких-либо действий внутри блокчейна. Это происходит за счет копирования структуры на все узлы (компьютеры) системы, что позволяет каждому участнику иметь достоверную информацию обо всех транзакциях без какой-либо необходимости получать её из централизованного источника. Грубо говоря, не может быть такого, что кто-то из пользователей подделает транзакцию, какие-либо данные в сети, потому что все изменения проходят через проверки и подтверждения сразу в нескольких узлах сети. Так же прозрачность сети обеспечивает строгая иерархия блоков, где каждый следующий блок ссылается на хеш предыдущего блока.

База данных растёт до тех пор, пока узлы продолжают предлагать и подтверждать новые блоки. Целостность и подлинность попадающих в блокчейн транзакций гарантируется с помощью криптографических техник, позволяющих эффективно выявлять любые некорректные, искусственные транзакции. Эта «магия» скрыта от пользователя, но делает работу блокчейна возможной. Благодаря высокому уровню безопасности, блокчейн используется для защищенного хостинга приложений, хранения различных данных и легкой передачи цифровых финансовых инструментов, являющихся эквивалентом реально существующих денег.

В таких сетях как Bitcoin и Ethereum криптография используется для создания единой защищенной вычислительной среды из сотен тысяч машин, которая работает без единого владельца и какого-либо децентрализованного управления.

* 1. Основы безопасности в блокчейне.
     1. Введение в криптографию

Криптография - это наука, изучающая способы сокрытия данных и их конфиденциальности. Самый простой пример – шифр Цезаря, где каждый символ алфавита заменялся на символ n позиций дальше. До 1976 года криптография представляла собой шифровальный метод с одним секретным ключом, который предоставлял доступ к расшифровке данных. Дальше были разработаны методы с открытым ключом. И с этого момента криптография получила более широкое распространение и стала применяться частными лицами и коммерческими организациями, а в 2009 году была выпущена первая криптовалюта Биткоин.

Шифрование — один из наиболее важных инструментов, используемых в криптографии. Это средство, с помощью которого сообщение превращается в нечитаемый набор символов, если его непреднамеренно кто-то прочитает. Только отправитель и получатель знают, что скрыто в письме.

В современных технологиях широко используются три формы шифрования:

* симметричная криптография
* асимметричная криптография
* хеширование

**Симметричная криптография** подразумевает такой принцип работы: мы хотим передать сообщение A, шифруем его с помощью ключа K, передаем зашифрованный текст B другому пользователю, он с помощью всё того же ключа K расшифровывает сообщение B и получает исходные данные A.

ENCRYPTION

DECRYPTION

Encrypted text

ABC

ABC

Используем один и тот же ключ

K

K

1. Симметричная криптография

Так же симметричное шифрование можно разделить на два блока - блочные шифры и потоковые шифры.

Использование потоковых шифров подразумевает использование фиксированного ключа, который заменяет сообщение псевдослучайной строкой символов. Каждый символ шифруется по очереди, по одному биту.

Блочное шифрование позволяет делать длину ключа не равной длине сообщения, что позволяет зашифровать длинный текст.

Рассмотрим теперь плюсы и минусы симметричной криптографии. Главное её достоинство – она не требует больших средств. Но есть очень существенный недостаток – злоумышленник может перехватить ключ и соответственно расшифровать все нужные ему данные.

Соответственно необходимо было создать более безопасный способ шифрования данных. в 1970 году британский математик и инженер Джеймс Эллис дошел до концепции, что шифрование и дешифрование - обратные операции на основе двух разных ключей: приватного и публичного. Идея была построена на двух принципах:

Одностороння функция с потайным входом: можно перейти из первого состояния во второе, но обратно нельзя. Предположим, K – публичный ключ, k – приватный ключ. При этом, ключи математически связаны друг с другом через определенную функцию K=f(k).

Второй принцип – протокол Диффи-Хеллмана, позволяющий при использовании незащищенного канала связи получить двум сторонам секретный ключ для расшифровки данных.

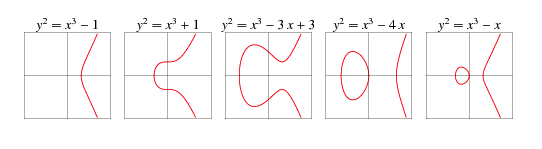
* + 1. Эллиптические кривые (ECC)

В криптовалютах Bitcoin, Ethereum и многих других используется криптография на эллиптических кривых (*Elliptic curve cryptography, ECC*). Так же системы на эллиптических кривых используются в TLS, PGP, SSH, важнейших технологиях на которых базируется современный веб и мир IT.

Внешне функция эллиптической кривой довольно простая:

При этом (исключаем особые кривые).

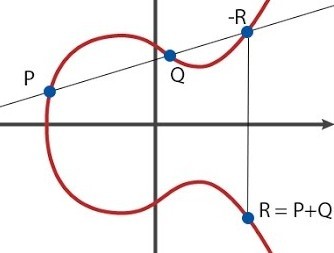
Изменяя параметры a и b график выглядит по-разному:



1. Графики эллиптических кривых

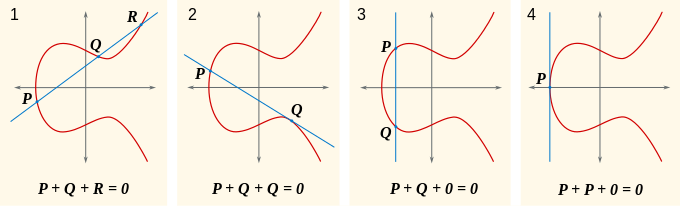
Как же эллиптические кривые помогают шифровать данные?

Рассмотрим некоторую эллиптическую кривую α. Две точки лежат на кривой P, Q. Их суммой называется точка R ∈ α, которая определяется следующим образом: проведем прямую через P и Q, она пересечет кривую в точке –R, поменяем *y* координату на противоположную по знаку и получим точку R. Это и есть та самая сумма P + Q = R:



1. Точки на эллиптической кривой

Важно заметить, что могут быть такие случаи, когда третье пересечение прямой через P или Q с кривой отсутствует. Или, например, прямая параллельна оси ординат. Тогда введем так называемую точку в бесконечности (point of infinity) *O* или *0*. И будем считать, что в случае отсутствия пересечения P + Q = 0.



1. Различные случаи пересечения кривой

Одно из правил сложения точек на эллиптической кривой – сумма трех точек на одной прямой (P, Q, R) равна 0. Эти точки образуют абелеву группу.

Определим операцию **скалярного умножения** и **логарифмирования (деления).** Примечательно, что из соотношения Q = n\*P узнать Q, зная n и P, легко. А обратный процесс по поиску n это сложная задача логарифмирования. И это, пожалуй, самое главное свойство эллиптических кривых.

Попробуем разобраться во всем этом. Ограничим кривую конечным полем – множество целых чисел по модулю p (p – простое число). Получается, что . Вместо непрерывной кривой мы получаем набор точек на плоскости.

Подробнее стоит остановиться на скалярном умножении. Тут вводится такое понятие, как циклическая подгруппа. Когда мы берем точку P кривой, мы можем найти другие точки, кратные P, которые циклически повторяются (0, P, 2P, 3P, 4P, 5P, 6P…). Также полученный набор точек циклически замкнут (т. е. результатом сложения между ними всегда будет какая-то из этих же точек). Получается, при сложении двух значений, кратных P, получаем значение кратное P. Точку P называют генератором или базовой точкой какой-то конкретной циклической подгруппы.

Циклические подгруппы – фундамент ECC и других криптосистем.

Определим еще несколько понятий. Порядок эллиптической кривой – количество точек в ней. Порядок циклической подгруппы – количество точек в этой подгруппе. В кривой может быть несколько циклических подгрупп, и порядки этих подгрупп будут являться делителями порядка исходной группы.

Пример: дана кривая , поле , это поле имеет порядок 42 (количество точек в поле). Подгруппы кривой могут иметь следующий порядок: n = 1, 2, 3, 6, 7, 12, 21. Подставим, например, точку P = (2, 3), посчитав, увидим, что P ≠ 0, 2P ≠ 0, …, 7P = 0, то есть порядок P равен n = 7.

Для осуществления алгоритмов ECC требуются подгруппы с высоким порядком. Поэтому берут эллиптическую кривую, вычисляют ее порядок, находят наибольший делитель, и затем находят подходящую базовую точку.

Следующий вопрос, который необходимо обсудить – дискретное деление. Повторим, надо узнать каким будет k, если мы знаем P и Q, и должно выполняться Q = k \* P.

Такая задача аналогична подобной в других криптосистемах (DSA, D-H протокол). Только в тех задачах используется не скалярное умножение, а возведение в степень по модулю.

Задача в ECC является более сложной по сравнению с другими схожими задачами, используемыми в криптографии. Это подразумевает, что нам потребуется меньше бит для целого k, чтобы получить тот же уровень защиты, что и в других криптосистемах.

Теперь приступим к рассмотрению алгоритмов, построенных на принципе эллиптических кривых.

* + 1. Алгоритмы криптографии в блокчейн

**Приватный ключ**

Одно из основополагающих понятий в криптографии – приватный ключ. В сети Bitcoin, как и в Ethereum используется алгоритм ECDSA.

В этом случае приватный ключ представляет из себя натуральное число от 1 до 2256. На данный момент перебрать все значения из этого диапазона практически невозможно. Это фантастически большое число.

**Открытый ключ**

Допустим k – число, наш приватный ключ. G – базовая точка на эллиптической кривой. Тогда внешний ключ можно вычислить путем перемножения G на k. Фактически получается, что публичный ключ — это точка лежащая на кривой.

Операция получения публичного ключа определена однозначно, как несложно увидеть из соотношения K = G \* k. То есть приватному ключу всегда соответствует только один публичной ключ.

Как мы уже успели разобрать, операция получения по внешнему ключу приватного является вычислительно трудной. И в общем случае получить приватный ключ из публичного можно только полного перебора всех значений от 1 до 2256.

Точно такая же связь существует между публичны ключом и адресом, только там всё дело заключается в необратимости хэш-функций.

Приватный ключ

k

Публичный ключ

Адрес

K

A

ECC

Хэш функция

1. Схема перехода от приватного ключа к адресу

Подытожим, в сети блокчейн используется уравнение кривой, поле значений на кривой, базовая точка на кривой и порядок базовой точки:

**Кривая**: на **поле** где Fp равно = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F.

**Базовая точка**: 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8.

Подчеркнута координата X в шестнадцатеричной записи, следом за ней идет координата y.

**Порядок базовой точки**: FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141.

Такой набор известен как SECP256k1. Такая кривая используется совместно с ECDSA (алгоритм цифровой подписи)

**ECDSA**

В данном алгоритме приватный ключ – это случайное число между 1 и значением порядка. Публичный ключ получаем путем умножения приватного на значение базовой точки.

Следовательно, максимальное количество секретных ключей (и адресов тоже) равно порядку. Но это невероятно большое число. Как мы уже сказали вычислить публичный ключ по секретному легко, а вот обратный процесс почти невозможен.

Когда у нас есть пара публичный ключ/приватный, ее можно использовать для подписи данных. Эти данные могут быть любой длинны. Сначала мы выполняем хеширование данных чтобы получить уникальное значение с числом битов равным битности порядка кривой.

После процесса хеширования алгоритм выглядит следующим образом:

* Выбирается k в диапазоне от 1 до n-1 (n – порядок)
* Рассчитывается точка k\*G.
* Находим r = x mod n. Если r получили равным нулю, то возвращаемся к шагу 1.
* Находим s = (x + r\*d)/k mod n. Если получили 0, то возвращаемся к 1 шагу (d – секретный ключ).
* Мы получили пару r и s – это и есть та самая подпись.

Следующее что нам надо сделать, это расшифровать полученные данные, не зная при этом секретного ключа. Q – публичный ключ. Порядок действий таков:

* Проверяем что r и s в диапазоне от 1 до n - 1.
* Считаем w = s-1 mod n, u = z\*w mod n, v = r\*w mod n.
* Рассчитываем точку (x,y) = uG + vG.
* Если r = x mod n, то подпись истинна.

В самом деле: u\*G + v\*Q = u\*G + v\*d\*G = (u + v\*d)\*G = (z\*s-1 + r\*d\*s-1)G = (z + r\*d)\*s-1\*G = k\*G.

Безопасность ECDSA основана на сложности поиска секретного ключа. Также безопасность зависит от выбора k в процессе подписи данных.

* + 1. Уязвимости блокчейна

Все уязвимости и возможные атаки на блокчейн сеть можно подразделить на несколько категорий:

* воздействие на уровне сети
* воздействие на уровне пользователя
* воздействие на уровне майнинга
* атаки, не зависящие от блокчейна
  + - 1. Уязвимости на уровне сети
* **DDoS атака** (Distributed Denial of Service).

Заключаются в отправке большого количества вредоносных файлов на потенциально уязвимый, слабо защищенный узел, обрабатывающий в данный момент транзакцию. Далее вредоносный код может передаваться из узла в узел, пока не заразит всю сеть.

На данный момент разработано множество средств защиты от таких атак. Но данный тип атаки всё же очень опасен.

Например, в марте 2018 года сеть платежных каналов второго уровня для блокчейна биткоина [Litghtning Network](https://lightning.network/)[подверглась](https://twitter.com/alexbosworth/status/976158861722726405?ref_src=twsrc%5Etfw&ref_url=https%3A%2F%2Fru.insider.pro%2Ftopnews%2F2018-03-22%2Fna-set-lightning-network-sovershena-ddos-ataka%2F&tfw_site=InsiderPro_Ru) DDоS-атаке. Из строя было выведено больше 20% узлов.

* **Атака Сивиллы.**

В одноранговых сетях, таких как Bitcoin и Ethereum, нет доверенных нодов, поэтому каждый запрос пересылается нескольким получателям. В тот же момент пользователи могут иметь несколько идентификаторов из разных нодов, которые можно использовать для разделения общих ресурсов. Полученные копии создают избыточность, позволяют проверять принятые независимые из сети данные.

Но если смотреть на этот подход с другой стороны, то выходит так, что все доступные ноды, которые должны представлять разных получателей запроса, контролируются одним и тем же пользователем. Если он окажется мошенником, то следующие транзакции замкнутся на нодах-псевдонимах.

Данная атака набирает популярность, ведь децентрализованная сеть растет, и при большом количестве пользователей нецелесообразно требовать от каждого участника сети подтверждать владение своими идентификаторами, ведь это препятствует масштабируемости сети.

* **Eсlipse attack (**атака информационного затмения**).**

Данный тип атаки применим как к сети Bitcoin, так и к Ethereum.

Атака позволяет получить контроль к узлу и информации в нём. При определенных навыках злоумышленник может заставить узлы контактировать только с зараженными узлами.

* **Взлом алгоритмов хеш-функций**

Почти невозможно вычислить хэши стандартов SHA-256 и ECDSA. Но при появлении квантовых компьютеров ситуация немного поменялась. Эта задача для них стала реальной. И риск взлома сложнейших алгоритмов увеличился.

* + - 1. Уязвимости на уровне пользователя
* **Ботнеты**

Для майнинга требуются огромные вычислительные мощности. Для экономии энергии преступники заражают компьютеры ничего не подозревающих пользователей вредоносным ПО. А именно, скрытыми майнерами.

Так, ботнет под названием Smominru для майнинга Monero за полгода [заразил](https://www.zdnet.com/article/a-giant-botnet-is-forcing-windows-servers-to-mine-cryptocurrency/) более полумиллиона серверов по всему миру и принес 8900 XMR, или $2 миллиона.

Уязвимость на уровне пользователя с юридической точки зрения связана с деанонимизацией участников рынка. Так как блокчейн-адреса не привязаны к личности и все проводимые транзакции не требуют раскрытия участников сделки, крипто-мошенники начинают пользоваться этими преимуществами в своих корыстных целях. Если злоумышленник подключит зараженные ноды к сети, то появится возможность проследить источник совершения транзакций.

* + - 1. Уязвимости на уровне майнинга
* **Атака 51%**

Атака заключается в том, что мошенник, контролируя более 50 процентов подтверждающих ресурсов сети блокчейн, может напечатать свою цепочку блоков, которая в результате станет основной. При этом у него будет возможность отменить транзакции в отброшенных блоках.

Такая атака бессильна перед сетями с большим количеством участников. Например, чтобы таким образом взломать BitCoin потребуется мощность во много раз превышающая суммарную мощность всех суперкомпьютеров мира.

Однако, ошибки в коде могут принести много проблем. Совсем недавно злоумышленники атаковали Verge. Проблема затронула все пулы и всех майнеров, из-за чего происходило отторжение всех корректных транзакций. Ошибка в коде позволила воспользоваться уязвимостью «51 процента» и похитить около 2 миллионов долларов.

* **Double spending**

Эта уязвимость позволила в 2013 году одному майнеру дважды потратить монеты на сумму 10000 долларов. Хоть в bitcoin и предусмотрен механизм защиты от повторных транзакций, но он дал сбой из-за конфликта разных версий клиентских программ Bitcoin.

* **Selfish mining**

Данная уязвимость пока не раскрыла себя в полной мере, но может значительно повлиять на развитие блокчейн индустрии в будущем. Причина этому – объединение мощностей организованной группы майнеров с целью получить контроль над сетью.

Например, сейчас большое количество майнеров находится в Китае. Именно эта страна добывает две трети всех биткоина мира. Если данный процесс будет набирать обороты, то децентрализованная сеть превратится в централизованную, что категорически разрушает принцип построения сети.

* **Замедление времени в системе**

Сценарий этой атаки такой, хакеры атакуют сеть в которой находятся абоненты блокчейн продукта, к примеру, биткоина, и путем создания большой вычислительной нагрузки на систему замедляют время внутри сети, что усложняет передачу данных, сообщений между пользователями, обновление информации в сети, формирование блоков, цепочек и их фиксацию участниками транзакций.

* + - 1. Уязвимости, не зависящие от блокчейна
* **Фишинг**

За 2017 год более 50% средств из блокчейн проектов было украдено при помощи фишинга. Это примитивная атака, когда человек переходит по вредоносной ссылке, передавая свои пароли, приватные ключи и все другие необходимые для взлома данные. Так же злоумышленники подделывают сайты блокчейн проектов. Например, в апреле 2018 года хакеры похитили с адресов кошелька MyEtherWallet около 150 миллионов долларов благодаря фишингу.

* **Дефейс**

Это процесс взлома сайта блокчейн проекта с последующей сменой адресов сбора средств на те, которые нужны злоумышленникам. Например, в июле 2017 года CoinDash подвергся данному типу атаки, потеряв при этом около 40000 монет Ethereum, что приблизительно равно 7 миллионам долларов.

1. Уязвимости смарт-контрактов

Одна из самых важных проблем – организация безопасности в смарт-контрактах на платформе Ethereum. Смарт-контракты в последнее время широко распространены, поскольку правильно написанный контракт обеспечивает широкий функционал. Кроме того, использование приложений на платформе Ethereum дешевле и безопаснее, многих стандартных систем. Конечно, при правильном подходе к разработке.

Про неправильный подход к разработке я хотела бы рассказать, сконцентрировав своё внимание ан проблему уязвимости смарт-контрактов.

Смарт-контракты выполняются в точности так, как их запрограммировали. А из-за того, что это довольно новое направление в IT, программируют их в большинстве случаев, совершая множество ошибок.

Все проблемы в разработке можно разделить на несколько категорий:

* Проблемы в коде
* Проблемы в процессе разработке
* Проблемы в языке Solidity
* Проблемы в концепции

Рассмотрим каждую подробнее.

* 1. Уязвимости в коде контракта

Уязвимости в коде могут быть самые разные. На безопасность контракта оказывают влияние самые разные факторы написания кода. Например, неправильный выбор архитектуры контракта. Не желательно в одном контракте пытаться реализовать кучу функций, это приведет к сложностям с поддержкой кода контракта, аудирования и модифицирования его.

Хорошая новость, что такие проблемы можно выявить и устранить на стадии разработки.

* + 1. Re-entrancy

Другое название – рекурсивный вызов. Одной из особенностей платформы Ethereum является то, что контракты могут в процессе своей работы вызывать и выполнять код других смарт-контрактов. Когда контракт обращается к другому контракту происходит вызов fallback-функции. А эта функция может содержать любой код, который злоумышленник захочет туда поместить.

Проблема заключается в том, что уязвимый контракт совершает вызов к другому контракту, и при этом внешний контракт может делать вызов функций уязвимого контракта внутри начального вызова. Поэтому данная уязвимость и называется рекурсивной.

Чтобы исследовать данную ошибку я создала уязвимый контракт MyVuln. Всю работу я проделывала в среде Remix IDE. Она предлагает широкий спектр возможностей по отладке контракта, его разворачиванию в тестовой сети Ethereum.

Фрагмент кода из MyVuln.sol:

function withdrawSomeMoney(uint \_someMoney) public {

require (\_someMoney <= balances[msg.sender]);

require(msg.sender.call.value(\_someMoney)());

balances[msg.sender] ‐= \_someMoney;

}

Добавим атакующий контракт MyXaker.sol. Фрагмент кода из него:

MyVuln public vuln;

function withdrawFromVuln() {

vuln.withdrawSomeMoney(100);

}

function () payable {

vuln.withdrawSomeMoney(100);

}

При вызове функции *withdrawFromVuln*, вызывается функция *withdrawSomeMoney* из атакуемого контракта. Контракт отправляет средства функцией *msg.sender.call.value().* Но этой функцией вызывается fallback функция контракта *MyXaker*. В ней прописан код на новый вызов функции вывода средств с уязвимого аккаунта. Но с баланса средства еще не успели списаться, поскольку мы пока не дошли до строчки *balances[msg.sender] ‐= \_someMoney*. То есть проверка достаточности баланса успешна, и мы снова попадаем на fallback функцию контракта злоумышленника. И так происходит до тех пор, пока на аккаунте MyVuln совсем не останется средств.

Эксплойт стал причиной краха DAO. Убыток от этой уязвимости оценивается в 50 миллионов долларов.

Как обойти эту уязвимость? Безопасно перевести средства можно с помощью функции transfer, поскольку она ограничивает исполнение кода до 2300 газа. Этого количества газа недостаточно чтобы исполнить повторный вызов контракта из fallback-функции атакующего контракта.

И всегда надо помнить, что любая функция, исполняющая внешний код, несёт в себе опасность.

* + 1. Access control

Контроль доступа в переводе на русский язык. Такая уязвимость может позволить злоумышленнику стать владельцем контракта или заставить пользователя авторизоваться в необходимом злоумышленнику контракте.

В языке solidity существуют привычные нам модификаторы функций: private, public, external, internal. Использование неподходящих спецификаторов может привести к неблагоприятным последствиям.

Для исследования этой уязвимости я написала контракт кошелька Wallet, позволяющий хранить владельцу контракта эфир на нем, с возможностью вывода всего баланса контракт на указанный адрес. Приведу в пример функцию вывода средств с кошелька:

function withdrawAll(address payable \_luckyAddress) public {

require(tx.origin == owner);

\_luckyAddress.transfer(balance);

}

Создадим также атакующий контракт *Xaker*. И fallback функция из него:

function () external payable {

poorWallet.withdrawAll(attacker);

}

Я протестировала вполне реальную ситуацию: на балансе контракта Wallet хранится сумма 4000 эфира. Потом я развернула контракт *Xaker*, указав при этом адрес уязвимого контракта и адрес своего второго кошелька, «злоумышленника». Затем с адреса «потерпевшего» я перечислила 1 ether на счет контракта *Xaker*. Что же произошло дальше?

Баланс злоумышленника увеличился на те самые 4000 ether. А в контракте Wallet не осталось средств. Как это произошло?

Чтобы ответить на этот вопрос надо присмотреться к функции *withdrawAll* в *Wallet*. Когда потерпевший перечислил на контракт *Xaker* эфир выполнилась fallback функция, в которой выполнен вызов функции вывода всех средств на счет злоумышленника из контракта Wallet. В кошельке стоит проверка *require(tx.origin == owner)*, определяющая кто обращается к функции. И тут важно заметить, что *tx.origin* – это глобальная переменная в Solidity, определяющая значение адреса исходного аккаунта, вызывавшего функцию или отправившего транзакцию. И в нашем случае это адрес владельца контракта Wallet. Таким образом проверка успешно проходит и весь баланс кошелька отправляется прямо к злоумышленнику.

Способ защиты от такой атаки – использование вместо *tx.origin* переменной *msg.sender*. Она уже определяет непосредственно вызвавшего функцию пользователя.

* + 1. Unchecked return values for low level calls

В переводе – отсутствие проверки возвращаемых значений низкоуровневых вызовов. Одной из главных особенностей языка программирования Solidity является наличие низкоуровневых функций. Это такие функции, как *call*, *send*, *callcode*, и *delegatecall*.

Их поведение в случае возникновения ошибок сильно отличается от таких функций, как например, *transfer*. В случае какой-либо ошибки они возвращают значение false, но сам код после этих функций продолжает выполняться, а отличие от *transfer*. Также *transfer* еще и откатывает транзакцию. Возвращаемое значение низкоуровневой функции необходимо всегда проверять.

Данную уязвимость можно рассмотреть на примере следующего кода:

function withdrawMoney(uint \_amount) public

{

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

msg.sender.send(\_amount);

balances[msg.sender] -= \_amount;

}

В данном коде нет никакой проверки на успешность операции перевода эфира. Получается, что в случае ошибки, например, недостаточного количества газа, пользователь не получит свои средства на счет, а в смарт-контракт запишется то, что он их уже получил.

Какие же ошибки могут возникнуть при отправке эфира из смарт-контракта на чей-то внешний счет?

Первое, это то что адрес пользователя – контракт, а код этого контракта генерирует исключение. Чтобы избежать данной уязвимости, рекомендуется использовать функцию *transfer*.

Второй случай менее очевиден. Виртуальная машина Ethereum EVM имеет ограниченный ресурс *callstack* (это глубина стека вызовов). И этот ресурс может быть использован другими кодом контракта, который был выполнен ранее в транзакции. То есть, если *callstack* уже израсходован к моменту выполнения команды send, выполнение команды потерпит неудачу. Деньги будут потеряны не по вине пользователя.

Как можно избежать данной ошибки? Передача при использовании send завершается с ошибкой, если глубина стека вызовов составляет 1024. Чтобы избежать проблем можно проверять возвращаемое значение send, чтобы убедиться, успешно ли оно завершено.

Добавим условие к нашему примеру

if(msg.sender.send(\_amount))

balances[msg.sender] -= \_amount;

else throw;

В данном примере мы всё сделали правильно и устранили проблему. Но я решила немного модифицировать наш пример:

if(msg.sender.send(\_amount) && owner.send(10)) {

balances[msg.sender] -= \_amount;

balances[owner] -= 10;

}

else throw;

}

Данный пример отличается тем, что деньги отправляются не одному пользователю, а сразу двум. Мы избавились от атаки *callstack* для sender-а. Но теперь sender и owner уязвимы друг к другу. В этом случае продолжается выполнение, если команда send по какой-либо причине дала сбой.

Поэтому лучшая защита – это проверка наличия ресурса *callstack*. Мы можем определить метод *callStackIsEmpty*(), который вернет ошибку если только *callstack* пустой. Тогда мы получим:

if (callStackIsEmpty()) throw;

При разработке смарт-контрактов лучше всего пользоваться функцией *transfer*. Но если неизбежно нужно использовать одну из низкоуровневых функций, то обязательно надо делать проверки на возвращаемые значения.

Разработчики об этом часто просто забывают и среди контрактов, использующих функцию send, больше половины не содержат абсолютно никакие средства защиты.

* + 1. Denial of service

В переводе – отказ в обслуживании. К этой проблеме можно отнести сразу несколько уязвимостей – достижение предела газа, нарушение контроля доступа, неожиданный крах.

Это чрезвычайно болезненные для индустрии смарт-контрактов на платформе Ethereum уязвимости. В то время как приложения другого типа могут в итоге восстановить все потерянные данные, исправить ошибки, восстановиться, смарт-контракты могут быть отключены и потеряны раз и навсегда всего лишь одной из таких атак.

К примеру, была популярная пирамида на блокчейне GovernMental. Смысл такой – есть таймер, когда он достигает нуля, весь джек-пот получает кто-то один. Проблема была в том, что участников в пирамиде было очень много, и для покрытия всех расходов нужна была сумма газа 5057945, а в контракте была заложена сумма в 4712388 газа. Таким образом, около 1100 эфира находится в подвешенном состоянии и с ними ничего нельзя сделать.

Рассмотрим для примера смарт-контракт со следующей функцией:

function give\_presents(address[] \_winners) public {

uint len = \_winners.length;

for (uint i=0; i<len; i++) {

require(\_winners[i].send(10));

}

}

В этом коде уже отсутствует уязвимость unchecked send, благодаря использованию проверки require. Но присутствует другая уязвимость. Если один победитель не сможет получить свой выигрыш по каким-либо причинам, то остальные тоже его не получат.

* + 1. Bad randomness

В переводе значит проблема случайных чисел. Так же эту уязвимость еще называют nothing is secret. Блокчейн Ethereum основан на детерминированном алгоритме. Поэтому получение случайных чисел – задача весьма трудная. Фактически, невозможно получить реально случайное число. Поэтому задача разработчика состоит в том, чтобы придумать алгоритм такой сложности, чтобы его было максимально сложно взломать.

Нельзя использовать в качестве источника псевдослучайного числа следующие элементы:

* переменные контракта, даже если они отмечены спецификатором доступа private
* блочные переменные

Это такие переменные, как block.coinbase (адрес который добыл текущий блок), block.difficulty (относительная мера того, насколько сложно было найти блок), block.gaslimit (эта переменная ограничивает максимальное количество газа для проведения транзакций внутри блока), block.timestamp (эта переменная обозначает время, когда был добыт блок). Все эти переменные могут отслеживаться кем угодно. Также переменные блока используются в одном и том же блоке. Таким образом, если контракт злоумышленника вызывает контракт жертвы посредством внутреннего сообщения, то один и тот же генератор случайных чисел даст одинаковый результат.

* хеши предыдущего и даже следующего блоков.

Каждый блок в блокчейн Ethereum имеет проверочный хеш. Виртуальная машина Ethereum позволяет получить такие блочные хеши с помощью функции block.blockhash(). Аргумент этой функции – число, указывающее на номер блока.

Более надежные способы генерации псевдослучайных чисел – использование внешних источников, алгоритм Signidice, подход Commit – Reveal.

* + 1. Time Manipilation

В Solidity переменная now соответствует глобальной переменной block.timestamp, на которую большое влияние оказывают майнеры. Рассмотрим к примеру следующий контракт:

contract Auction {

uint jackpot = 100 ether;

uint public lastBid = 1 ether;

constructor() public payable {}

function setBid() public payable {

require(msg.value >= lastBid);

if (now%20 == 0){

msg.sender.transfer(jackpot);

}

}

}

Этот контракт – определённая лотерея, в которой выигрывает тот, кто прислал ставку в момент, удовлетворяющий определенному условию (now%20 == 0). Для гарантированного выигрыша майнер может установить подходящее значение *block*.*timestamp* и смайнить блок с такой выигрышной транзакцией.

* + 1. Short Address Attack

В сети Ethereum для вызова контракта используется ABI смарт-контракта (Application Binart Interface). Это бинарный интерфейс приложения, согласно которому вызов функции представляется в виде последовательности байтов. Первые 4 байта – это начало хэш функции keccak256 от подписи функции (ее название и типы входных данных). Далее идут значения входных параметров.

Адрес в Ethereum состоит из 20 байт. Для осуществления атаки злоумышленнику необходимо сгенерировать адрес, который заканчивается на нулевой байт. Например, abcdabcdabcdabcdabcdabcdabcdabcdabcdab00.

В обычном случае вызов функции на перевод 1 ether будет выглядеть так:

a9059cbb|000000000000000000000000abcdabcdabcdabcdabcdabcdabcdabcdabcdab00|0000000000000000000000000000000000000000000000000000000000000001

Но если указать адрес без последнего байта, то недостающий байт адреса будет взят из следующего аргумента, а в конце EVM допишет нулевой байт:

a9059cbb|000000000000000000000000abcdabcdabcdabcdabcdabcdabcdabcdabcdab|000000000000000000000000000000000000000000000000000000000000000100

Получается, что вместо одного Ether жертва отправит 0x100 = 256 ether. Защититься от такой атаки поможет только тщательная проверка параметров перед загрузкой их в блокчейн.

* 1. Проблемы в процессе разработки

Проблемы в коде обусловлены в первую очередь неправильно выстроенным процессом разработки. Казалось бы, разработка ПО — дело давно изученное. Тем не менее, молодость области смарт-контрактов, непропорционально большие деньги приводят к тому, что люди пренебрегают стандартными процедурами, что часто приводит к серьёзным проблемам. Из самого типичного стоит упомянуть:

* **отсутствие ТЗ**
* **сроки**: в основном на разработку выделяется очень мало времени. Это служит причиной почему разработчики не успевают полностью протестировать свой код на уязвимости в оптимизации и безопасности.
* **уровень разработчика**: в область приходит много людей без какого-либо опыта в программировании. Это связано с чрезвычайно высокой оплатой разработки смарт-контрактов и отсутствии конкуренции в этой области.
* **понимание экосистемы сети**: для разработки контрактов важно понимать, как всё устроено изнутри, как работает EVM. Мало какие разработчики этим интересуются, а если и интересуются, то поверхностно.
  1. Уязвимости в языке Solidity

К проблемам в языке можно отнести множественное наследование, это значит, что код запускается не из текущего контракта, а значит снижается читаемость, а вследствие и безопасность кода.

Некоторые уязвимости я хотела бы рассмотреть подробнее:

* + 1. Неправильное использование DelegateCall

Это низкоуровневая функция, позволяющая исполнять методы стороннего контракта в контексте вызывающего контракта. Другими словами, при исполнении функции стороннего контракта используется хранилище вызывающего контракта. Таким образом, значения msg.call и msg.value остаются первоначальными.

По такому принципу была построена атака на кошелек Parity. В результате пользователь под никнеймом Devops199 смог стать фактически владельцем контракта и вызвать функцию самоуничтожения.

* + 1. Arithmetic Issues

Другими словами – уязвимость компьютерной арифметики, так же известна как целочисленное переполнение через нижнюю границу. Такая ошибка, как и в других языках, возникает из-за ограниченного размера памяти, выделенного на переменную. Например, максимальное значение для переменного типа uint равно 2256-1, минимальное - 0. Выйти за эти границы никак не получится.

Рассмотрим определенную функцию, которая используется во множестве реальных контрактов:

function batchTransfer(address[] \_receivers, uint256 \_value) public returns (bool){

uint cnt = \_receivers.length;

uint256 amount = uint256(cnt) \* \_value;

require(cnt > 0 && cnt <= 20);

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

balances[msg.sender] = balances[msg.sender].sub(amount);

for (uint i = 0; i < cnt; i++) {

balances[\_receivers[i]] = balances[\_receivers[i]].add(\_value);

Transfer(msg.sender, \_receivers[i], \_value);

}

return true;

}

Функция *batchTransfer* перечисляет эфир с заданным количеством в переменной \_*value* на адреса из массива \_*receivers*. Обратим внимание на строку вычисления *amount*. Мы умножаем количество адресов на количество эфира.

Зададим значение \_value равным 2256/\_receivers.length. Тогда у нас случится переполнение. И amount примет значение 0. Однако обе следующие проверки пройдут успешно, в том числе и на то что баланс отправителя больше amount. В итоге балансы получателей будут пополнены на величину \_value, если, например, получателей было два, то каждый из них получит по 2255 токенов. А баланс отправителя не поменяется. В следующих строках кода балансы двух получателей станут огромными.

Множество контрактов уязвимы к этой проблеме. Главный метод защиты от такой уязвимости – создание библиотеки безопасных математических операций. Я постаралась ее реализовать:

library MySafeMath {

//Сложение

function add(uint a, uint b) public constant returns(uint) {

uint c = a + b;

assert(c >= a);

return c;

}

// Вычитание

function sub(uint a, uint b) public constant returns(uint) {

assert(b <= a);

return a - b;

}

// Умножение

function mul(uint a, uint b) public constant returns(uint) {

uint = a \* b;

assert(a == 0 || c/a = b)

return c;

}

// Деление

function div(uint a, uint b) public constant returns(uint) {

uint c = a/b; // Деление на 0 EVM проверяет автоматически

return c;

}

}

Эту библиотеку в последствии можно дополнить и другими операциями. Очень важно использовать такие библиотеки при работе с реальными контрактами. Иначе можно стать жертвой хакеров.

* 1. Проблемы в концепции

Удивительно, но среди перечисленных мною проблем нет самоё главной – непонимание зачем вообще нужны контракты, что они могут и как они работают.

Это приводит к тому, что написанный **смарт-контракт**:

* **Далеко не смарт**: если он плохо написан, делает не то что задумано. Или если он сильно завязан на бэкэнде. В таком случае его выполнение полностью контролирует администратор.
* **И вовсе не контракт:** например, если контракт не фиксирует никаких обязательств, или допускает одностороннее изменение.
* **Не нужен:** на волне популярности стало модно в названия компании добавлять такие слова, как блокчейн, смарт-контракт. Но едва ли в 10 процентах случаев они действительно требуются. Больше это работает как всяческие попытки прикрутить новомодную технологию к чему-угодно, куда-угодно. Зачем?