Оглавление

[1. Введение 1](#_Toc6761985)

[1.1. Общие сведения о блокчейне 1](#_Toc6761986)

[2. Глава 1: безопасность в блокчейне 3](#_Toc6761987)

[2.1. Криптография в основе блокчейна 3](#_Toc6761988)

[2.1.1. Криптография на эллиптических кривых (ECC) 5](#_Toc6761989)

[2.1.2. Задача дискретного логарифмирования 9](#_Toc6761990)

[2.1.3. ECDH и ECDSA 9](#_Toc6761991)

[2.1.3.1. Случайные кривые 10](#_Toc6761992)

[2.1.4. ECDH 11](#_Toc6761993)

[2.1.5. ECDHE 12](#_Toc6761994)

[2.1.6. ECDSA 12](#_Toc6761995)

[2.1.7. Ethereum криптография на практике 13](#_Toc6761996)

[2.2. Уязвимости блокчейна 14](#_Toc6761997)

[2.2.1. Уязвимости на уровне сети 14](#_Toc6761998)

[2.2.2. Уязвимости на уровне пользователя 15](#_Toc6761999)

[2.2.3. Уязвимости на уровне майнинга 16](#_Toc6762000)

[2.2.4. Уязвимости, не зависящие от блокчейна 18](#_Toc6762001)

[3. Глава 2: Уязвимости смарт-контрактов 18](#_Toc6762002)

[3.1. Reentrancy 18](#_Toc6762003)

[3.2. Access control 21](#_Toc6762004)

[3.3. Arithmetic Issues 23](#_Toc6762005)

[3.4. Unchecked return values for low level calls 25](#_Toc6762006)

[3.5. Denial of service 28](#_Toc6762007)

[3.6. Bad randomness 29](#_Toc6762008)

[3.7. Front-running 30](#_Toc6762009)

[3.8. Time Manipilation 32](#_Toc6762010)

[3.9. Short Address Attack 33](#_Toc6762011)

1. Введение
   1. Общие сведения о блокчейне

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. Глава 1: безопасность в блокчейне
   1. Криптография в основе блокчейна

Что же вообще такое 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

Рассмотри подробнее каждую форму.

**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

Рисунок 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.

**Ассиметричная криптография**

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. Криптография на эллиптических кривых (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.

Давайте разберемся подробнее в эллиптических кривых.

Внешне это довольно простая функция. Как правило, записывается в форме Вейерштрасса:

При этом (исключаем особые кривые).

Изменяя параметры a и b график выглядит по-разному:

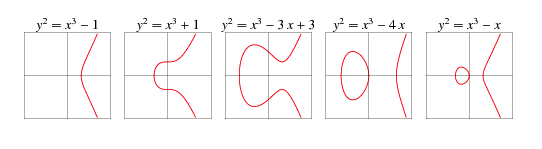


Рисунок 2 Графики эллиптических кривых

Впервые эта функция засветилась еще в трудах Диофанта, а позже и в исследованиях Ньютона, где он выводил формулы сложения точек на эллиптической кривой.

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:

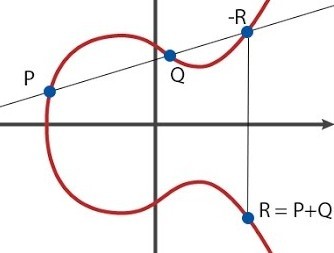


Рисунок 3 Точки на эллиптической кривой

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.

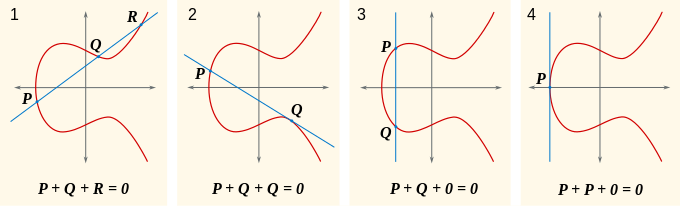


Рисунок 4 Различные случаи пересечения кривой

Важно вспомнить о правилах сложения точек на эллиптической кривой: сумма трех точек, лежащих на одной прямой равно 0, работают правила ассоциативности и коммутативности. Эти три точки образуют абелеву группу.

Определим еще одну операцию: **скалярное умножение** точки на эллиптической кривой. Это можно сделать с помощью операции удвоения-сложения. Приведем пример:

Число 151 можно представить как . Наша точка P. Следовательно запишем . Итак, алгоритм имеет следующий вид: берем P, удваиваем его, складываем 2P и P (чтобы получить ), удваиваем 2P (получим ), сложим с прошлым результатом (получим ), удваиваем (получим ), но в этот раз не производим сложения (нам не нужна 3 степень). Удвоим , получим , сложим с результатом и проделываем алгоритм дальше, пока не получим требуемое значение Q = 151\*P.

Определим операцию **логарифмирования (деления).** Мы узнали, как можно умножить точку на скаляр, но допустим нам необходимо проделать обратную операцию: из соотношения Q = n\*P узнать n. Это задача логарифмирования. И тут раскрывается, пожалуй, самое главное свойство эллиптических кривых – если уменьшить область определения кривых, скалярное умножение остается простой задачей, а вот логарифмирование (известное как дискретное) сложной.

Попробуем разобраться во всей этой магии. ограничим кривую не каким-то вещественным числом, а конечными полями. К примеру, возьмем поле – множество целых чисел по модулю p (p – простое число). Для полей существует две операции: сложение и умножение. Они обе замкнуты, ассоциативны и коммутативны. Для этих операций существует уникальный единичный элемент и для каждого элемента есть уникальный элемент обратной величины. Так же работает закон дистрибутивности умножения относительно сложения.

Рассмотрим несколько операций над полем :

* Сложение: (17+7) mod 23 = 1
* Вычитание (22 - 3) mod 23 = 4
* Умножение: 3\*8 mod 23 = 1
* Аддитивная инверсия: (5+18) mod 23 = 0
* Мультипликативная инверсия: ,

Ограничим эллиптические кривые полем . Получается, что . Вместо непрерывной кривой мы получаем набор точек на плоскости. Так же можно вычислять алгебраическую сумму точек, как и в случае непрерывной кривой.

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…). Но значения кратные P циклически повторяются (то есть другие точки никогда не становятся кратными P). 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 (исходная кривая содержит N точек, одна из подгрупп содержит n точек, то n является делителем N).

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. Введем понятия числа h = N/n – оно всегда целое и называется кофактором подгруппы.

Для каждой точки кривой есть NP = 0 (так как N кратно любому n), следовательно, можно записать n \* (h \* P) = 0. (h \* p) обозначим через G. Получим n \* G = 0 (получается, что точка G создает подгруппу порядка n).

Подытожив всё вышесказанное, получим алгоритм:

* Вычисляем порядок кривой N
* Выбираем порядок n подгруппы (число n простое и делитель N)
* Вычисляем кофактор h = N / n
* Выбираем случайную точку P
* Вычисляем G = h \* P (G не должно быть равно 0)

G – искомый генератор подгруппы порядком n и кофактором h.

* + 1. Задача дискретного логарифмирования

Следующий вопрос, который необходимо обсудить – дискретный логарифм (дискретное деление). Повторим, нам 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. Их задачу можно сформулировать следующим образом: если известны a и b, то каким будет k, такое что бы выполнялось уравнение:

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. ECDH и ECDSA

Алгоритмы работают в циклической подгруппе эллиптической кривой над конечным полем F. Нам требуются следующие параметры:

* p – размер конечного поля, простое число
* a и b – коэффициенты уравнения эллиптической прямой
* G – базовая точка, генерирующая подгруппу
* n – порядок подгруппы
* h – кофактор подгруппы (N/n)
* S – порождающее значение seed для генерации a и b или G
  + - 1. Случайные кривые

Задача дискретного логарифмирования сложна. Но не во всех случаях. Существуют некоторые классы эллиптических кривых, которые уязвимы к специальным алгоритмам. Например, все кривые, для которых *p = h \* n* уязвимы к атаке Смарта.

При генерации a и b, а также G мы используем хеш S. Хеши довольно просто вычислять, но очень сложно реверсировать этот процесс. Общая схема получения параметров a и b:

S = random ()

H = hash(s)

a = f(H)

b = g(H)

Рисунок 5 Общая схема генерации параметров a и b

Но получить хеш из параметров области определения почти невозможно. Придется решать задачу инверсирования хеша:

S

hash-1(H)

H = f-1(a)

H = f-1(b)

Рисунок 6 Получение хеша

Сгенерированная кривая называется проверяемо случайной. Такой принцип использования хешей известен как “nothing up my sleeve” (в моих рукавах ничего нет).

Таким образом, эта хитрость дает определенную гарантию, что кривая не была специально создана, чтобы иметь известные только ее автору уязвимости.

Стандартизированный алгоритм генерирования и проверки случайных кривых описан в ANSI X9.62 и основан на SHA-1.

Теперь перейдем к самому главному. Для безопасности в блокчейне используется закрытый и открытый ключ.

Закрытый ключ – случайное число d, выбранное из { 1, …, n – 1 } (n – порядок подгруппы).

Открытый ключ – это точка H = d \* G (G – базовая точка подгруппы).

Как мы видим, если мы знаем d и G, то найти H относительно легко. Но если мы знаем H и G, то поиск закрытого ключа d является сложной задачей дискретного логарифмирования.

На этом принципе основаны некоторые алгоритмы с открытым ключом: ECDH (Elliptic curve Diffie-Hellman, протокол Диффи-Хеллмана, используемый для шифрования) и ECDSA (Elliptic Curve Digital Signature Algorithm, используемый для цифровых подписей). Рассмотрим алгоритмы подробнее.

* + 1. ECDH

По-другому можно назвать протоколом согласования ключей. В сущности, это означает, что ECDH задает в определенной степени порядок генерирования ключей и обмена ими.

ECDH решает следующую проблему: две стороны хотят передать безопасно информацию, чтобы третья сторона мог перехватить её но не мог расшифровать.

Как это работает (рассмотрим на примере двух пользователей Алиса и Боб):

1. Сначала пользователи генерируют собственные открытые и закрытые ключи. Получаем, что у Алисы есть закрытый ключ и открытый ключ . Боб, в свою очередь, имеет закрытый ключ и открытый ключ . Важно заметить, что пользователи используют одну и ту же эллиптическую кривую, одно и то же конечное поле и одну и ту же базовую точку G.
2. Далее Алиса и Боб обмениваются открытыми ключами и по незащищенному каналу. Предположим, что посредник (MITM Man In The Middle) перехватил и . Но чтобы получить или необходимо решить почти нерешаемую задачу дискретного логарифмирования.
3. Алиса и Боб могут вычислить S с помощью собственного закрытого ключа и открытого ключа друг друга. и . S одинаковые для Алиса и Боба. Можно записать равенство:

Посреднику известны только и и он не сможет найти общий секретный ключ S без приватного ключа.

Протокол Диффи-Хеллмана как раз и заключается в том, что Алиса и Боб могут просто вычислить общий секретный ключ. А посреднику придется решать сложную логарифмическую задачу.

Получив общий секретный ключ, Алиса и Боб могут обмениваться данными с симметричным шифрованием. К примеру, они могут использовать координату x ключа S для шифрования сообщений такими безопасными шифрами, как AES и 3DES.

* + 1. ECDHE

Другими словами – эфемерное ECDH. Отличие в том, что передаваемые ключи временны, а не статичны.

* + 1. ECDSA

Задача заключается в следующем: Алиса хочет подписать сообщения своим закрытым ключом , а Боб хочет проверить подпись с помощью открытого ключа Алисы . Важно, что никто не может подделать подпись Алисы, и каждый сможет проверить подпись.

Для этого они снова используют одинаковые параметры области определения эллиптической кривой. Так же мы будем работать не с самими сообщениями, а с хешами сообщений, выбрав криптографическую хеш-функцию. При этом урежем длину хеш сообщения, сделав ее равной n (порядок подгруппы). Обозначим через *z* урезанный хеш.

Рассмотрим алгоритм подписывания сообщений:

1. Берем случайное целое k из {1, …, n - 1}. n – всё так же порядок группы.
2. Вычисляем точку P = k \* G.
3. Вычисляем число . – координата P.
4. Если r = 0, то выбираем другое k.
5. Вычисляем . Тут – закрытый ключ Алисы, – урезанный хеш, – мультипликативная инверсия k по модулю n.
6. Если s = 0, то выбираем другое k.
7. Получим пару r и s – это и является подписью.

Другими словами, хеш z Алиса подписывает с помощью закрытого ключа и случайного k. Боб проверяет правильность подписи сообщения с помощью открытого ключа Алисы . Проще говоря, мы генерируем ключ k, с помощью умножения точек секретный ключ прячется в r. Затем это самое r привязывается к хешу сообщения уравнением .

Важно помнить, что n должно быть простым числом.

Теперь рассмотрим процесс проверки подписи. Всё что у нас есть: открытый ключ Алисы , хеш z и сама подпись (r, s).

1. Вычисляем
2. Вычисляем
3. Вычисляем точку

Подпись действительна, если только

Необходимо хранить секретный ключ k в тайне. Потому что через k можно узнать закрытый ключ.

* + 1. Ethereum криптография на практике

Рассмотрим теперь всё это ближе к практике.

В Ethereum, как и в Bitcoin используется кривая [SECP256k1](https://en.bitcoin.it/wiki/Secp256k1) и алгоритм ECDSA. Она задается уравнением . И рассматривается на поле , где p – число равное . Приватный ключ в сети Ethereum – обычное число в диапазоне от 1 до 2256. Это чрезвычайно большое число, найти его методом перебора фактически невозможно. Публичный же ключ в сети, как мы уже выяснили, это произведение базовой точки на кривой на наш приватный ключ.

Соответственно каждому приватному ключу соответствует только один публичный ключ, то есть операция получения публичного ключа является однозначной. А обратная операция является математически трудной.

* 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 «мусорных» данных, вредоносных файлов на потенциально уязвимый, слабо защищенный узел, обрабатывающий в данный момент транзакцию. Сам пользователь даже не подозревает что его компьютер заражен и может передавать вирусные файлы на другие узлы цепочки в фоновом режиме.

На данный момент, к счастью, 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. Это доказало, что система требует серьезных доработок.

Данная 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

Одним из самых распространенных сейчас инструментов мошенничества является ботнеты.

* **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. Кроме того, анонимность блокчейна позволяет осуществлять сделки между террористами и другими преступниками, которые занимаются незаконной деятельностью.

* + 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.

На практике это выглядит так:

Появилась новая блокчейн сеть, в которую еще не успело прийти большое количество мощностей. Но тут приходит атакующий с суммарными вычислительными мощностями большими чем у остальных участников всей сети – он может не подтверждать чужие блоки, а подтверждать только свои, следовательно, получать 100% всех новых «биткоинов» и блокировать какие угодно чужие транзакции.

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 всего за несколько часов.

* **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. Один и тот же блок был подтвержден пользователями в версии 0.8, но не подтвержден в версии 0.7.

* **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 биткоинов 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

В принципе, следующие типы атак присущи любой сети. Но от этого их важность в блокчейне не уменьшается.

* **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. Глава 2: Уязвимости смарт-контрактов
   1. Reentrancy

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. Контракты часто отправляют эфир на внешние счета других пользователей. Эта операция требует отправки контрактом внешнего вызова. И когда контракт обращается к другому контракту происходит вызов fallback-функции (это функция, вызываемая при отправке на счет контракта эфира). 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 MyVuln1:

1. «Уязвимый к рекурсивному вызову смарт-контракт»:

contract MyVuln1

{

// *Балансы пользователей*

mapping (address => uint) private balances;

// *Внести средства*

function depositFunds() public payable

{

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

}

// *Вывести средства*

function withdrawSomeMoney(uint \_someMoney) public

{

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

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

balances[msg.sender] ‐= \_someMoney;

}

}

На первый взгляд никаких ошибок в коде нет, и контракт должен работать безопасно и надежно. Добавим атакующий контракт:

1. «Атакующий контракт»:

import "MyVuln1.sol";

contract MyXaker1

{

MyVuln1 public vuln;

function withdrawFromVuln()

{

vuln.withdrawSomeMoney(100);

}

function () payable

{

vuln.withdrawSomeMoney(100);

}

}

Мы вызываем функцию withdrawFromVuln, она в свою очередь вызывает функцию withdrawSomeMoney из атакуемого контракта. Контракт отправляет средства функцией msg.sender.call.value(). Но этой функцией вызывается fallback функция контракта MyXaker1. 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 функци контракта злоумышленника. И так происходит до тех пор, пока на аккаунте MyVuln1 совсем не останется средств.

Эксплойт стал причиной краха DAO. Убыток от этой уязвимости оценивается в 50 миллионов долларов. Reentrancy возникает, когда внешним вызовам разрешено совершать новые вызовы до завершения первоначального исполнения. Это означает, что состояние контракта может измениться в середине его выполнения.

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. 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.

Рассмотрим к примеру следующий контракт:

1. «Контракт кошелёк»

contract Wallet

{

address public owner;

uint public balance;

constructor (address \_owner) public

{

owner = \_owner;

}

function () external payable

{

balance += msg.value;

}

function withdrawAll(address payable \_luckyAddress) public

{

require(tx.origin == owner);

\_luckyAddress.transfer(balance);

}

}

Создадим также атакующий контракт:

1. «Контракт атакующий»:

contract Xaker {

Wallet poorWallet;

address payable attacker;

constructor (Wallet \_poorWallet, address payable \_attackerAddress) public

{

poorWallet = \_poorWallet;

attacker = \_attackerAddress;

}

function () external payable

{

poorWallet.withdrawAll(attacker);

}

}

Я протестировала вполне реальную ситуацию: на балансе контракта Wallet хранится сумма эфира, допустим 20. Также злоумышленник создал контракт Xaker, указав при этом адрес контракта Wallet и свой личный кошелек в конструкторе. Потом с помощью социальной инженерии злоумышленник заставил владельца кошелька перевести 1 эфир на контракт Xaker. После перевода все средства с контракта wallet перечислились на счет злоумышленника. Как это произошло?

Чтобы ответить на этот вопрос надо присмотреться к функции withdrawAll в Wallet. Когда потерпевший перечислил на контракт Xaker 1 эфир выполнилась fallback функция, в которой выполнен вызов функции вывода всех средств на счет злоумышленника из контракта Wallet. Но ведь 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.

Еще одна уязвимость из этой категории – 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.

Все средства после самоуничтожения фактически пропадают. К ним никак нельзя обратиться, они бесполезны, навсегда.

Однако сам атакующий утверждал, что он сделал это не специально, что он вообще новичок в этой теме. Убытки составили порядка 150-300 миллионов долларов.

Что можно сделать с потерянными деньгами? Ответ – ничего. Но есть некоторые варианты. Однако они очень спорные и неоднозначные.

После взлома Dao ситуация была разрешена с помощью хард-форка. Ethereum Foundation выпустили новую версию своего клиента, куда были внедрены новые транзакции, чтобы отобрать эфир из контролируемых хакерами контрактов. Формально, такие транзакции нарушили правила сети Ethereum. Многие пользователи не согласились с таким решением с аргументом, что “код – есть закон”. Поэтому они продолжали использовать оригинальное ПО.

Таким образом, возникло две версии Ethereum: Ethereum и Ethereum Classic. Старая версия существует и по сей день. Однако её капитализация на рынке примерно в 25 раз меньше новой версии Ethereum. Будет ли еще такие хард-форки? Навряд ли, учитывая разногласия, возникшие в прошлый раз.

* 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 (uint256 в другом обозначении) равно 2256-1, минимальное - 0. Выйти за эти границы никак не получится.

Я не буду рассматривать простые примеры, в которых арифметические действия выполняются без проверок. Рассмотрим определенную функцию, которая используется во множестве реальных контрактов:

1. «Контракт batchTransfer»

function batchTransfer(address[] \_receivers, uint256 \_value) public whenNotPaused 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;

}

Рассмотри этот контракт поподробнее. 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 очень большими, и для этого не потребуется ни копейки.

Данную уязвимость компания peckshield (они впервые обратили внимание на эту проблему) назвала batchOverflow. Оказалось, что множество контрактов уязвимы к этой проблеме.

Главный метод защиты от такой уязвимости – создание библиотеки безопасных математических операций. Я постаралась её реализовать и сделать максимально безопасной, определив операции сложения, вычитания, умножения и деления.

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;

return c;

}

}

Хотелось бы обратить внимание на метод деления. Мы не проверяем деление на 0, потому что сама EVM это делает.

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

* 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 функция transfer() наоборот прекращает выполнение кода и откатывает транзакцию. Это очень важно помнить при разработке смарт-контрактов на платформе Ethereum. Возвращаемое значение низкоуровневой функции необходимо всегда проверять, иначе это может привести к нежелательным результатам.

Данную уязвимость можно рассмотреть на примере следующего кода:

1. «Уязвимый контракт»

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, что может случиться, это то что адрес пользователя – контракт (а не учетная запись пользователя), а код этого контракта генерирует исключение (например, если он использует слишком много газа). 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?

В документации Ethereum есть некоторая информация по этому поводу. 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

1. «проверка send»

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:

1. «модификация примера»

function withdrawMoney(uint \_amount) public {

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

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:

1. «проверка callStackIsEmpty»

function withdrawMoney(uint \_amount) public {

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

if (!callStackIsEmpty()) {

msg.sender.send(\_amount);

owner.send(10);

balances[msg.sender] -= \_amount;

balances[owner] -= 10;

}

}

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 джекпот 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:

1. «Уязвимая функция»

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.

Ни в коем случае 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.

Всё потому что во всех этих вариантах псевдочисла могут быть предугаданы.

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

* 1. Front-running

В сети Ethereum все транзакции сначала добавляются в пул, который виден абсолютно всем участникам сети, а далее майнеры добавляют их в блок в произвольном порядке. Поэтому результат транзакции не должен зависеть каким-либо образом от ее положения в блоке. рассмотрим следующий контракт в качестве примера:

1. «Контракт GIFT»

contract GIFT {

bool passHasBeenSet = false;

address sender;

bytes32 public hashPass;

function () public payable {}

function GetHash(bytes pass) public constant returns(bytes32)

{

return sha3(pass);

}

function SetPass(bytes32 hash) public payable

{

if ((!passHasBeenSet && (msg.value > 1 ether)) || hashPass == 0x0)

{

hashPass = hash;

sender = msg.sender;

}

}

function GetGift(bytes pass) external payable

{

if (hashPass == sha3(pass))

{

msg.sender.transfer(this.balance);

}

}

function Revoce() public payable

{

if (msg.sender == sender)

{

sender.transfer(this.balance);

}

}

function PassHasBeenSet(bytes32 hash) public

{

if (msg.sender == sender && hash == hashPass)

{

passHasBeenSet = true;

}

}

}

Этот контракт задумывался как сервис отправки подарка другому пользователю. Подарком выступает эфир. То есть, первый пользователь кладет на аккаунт эфир с помощью метода SetPass, при этом задавая хэш SHA-3 вашего пароля (подразумевается, что он доступен получателю). Получатель же получает деньги методом GetGift.

Но в этом контракте кроется ловушка. Она находится в методе SetPass. В контракте есть переменная passHasBeebSet. Её нужно задавать отдельным одноименным методом. Однако этот метод не был вызван, и переменная до сих пор имеет значение false. Причем метод PassHasBeebSet может вызвать только тот, кто сначала вызывал метод SetPass.

То есть, теоретически, кто угодно может вызвать метод SetPass, с пополнением баланса больше чем на 1 эфир, и он станет sender-ом. Причем чтобы никто другой им уже не стал, необходимо сразу же вызвать метод PassHasBeenSet или просто вывести деньги с помощью методов Revoce/GetGift.

Всё вроде просто, но существует атака опережения. Смысл ее в следующем: злоумышленник наблюдает пул ожидающих транзакций и ждет вашу транзакцию. Как только она появляется в пуле, злоумышленник выполняет транзакцию с большей ценой газа. Транзакция злоумышленника пришла последней в текущем раунде, но благодаря наивысшей цене газа она в реальности будет исполнена раньше, чем ваша транзакция.

Конечно же, злоумышленник вызовет метод PassHasBeenSet. Это даст ему возможность избежать смены sender-а и, кроме того, все ваши деньги, отосланные методом SetPass, благополучно осядут в контракте. Ну а дальше он их просто выведет.

Хотелось бы отметить, что таких контрактов-ловушек очень много в сети Ethereum. И прежде чем куда-то отсылать свои деньги необходимо изучить код. Иначе вы рискуете потерять все свои средства. Эти ловушки основаны на разных типах уязвимостей в смарт-контрактах.

* 1. Time Manipilation

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

1. «Уязвимый к time manipulation 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.

Идем далее.

* 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.