Уязвимости смарт контрактов блокчейн платформы Ethereum

И. А. Алиев
Финансовый университет при Правительстве Российской Федерации (Финуниверситет), Financial University
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 важным является их безопасность относительно атак, нацеленных на кражу этих ресурсов или выведения 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

I. О смарт контрактах Ethereum

Блокчейн платформа Ethereum позволяет ее пользователям реализовывать на ней 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 (EVM), которая является основной этой блокчейн платформы.

A. EVM

В случае с платформой Ethereum программным обеспечением для узлов сети является Ethereum Virtual Machine. EVM можно назвать средой выполнения контрактов Ethereum, поскольку их код выполняется именно ею.

EVM работает не с кодом, написанном на языке высокого уровня, а с так называемым EVM байт кодом, в который предварительно компилируется исходный код контракта. Данный байт код представляет собой 16-теричную строку, которая является представлением кодов операций и операндов.

Каждой операции соответствует цена ее исполнения - так называемый *gas*. Эта особенность EVM – требование платы за каждую операцию – гарантирует защиту от DoS атак, являющихся наиболее популярным способом атаки в настоящее время.

B. Плата за исполнение кода

Для того, чтобы мотивировать узлы выполнять вычисления в сети Ethereum существует понятие платы за вычисления: каждая выполняемая операция имеет свою конкретную стоимость, выражающуюся в так называемом количестве *gas*'а.

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.

C. Структура смарт контракта

Смарт контракт, написанный на языке высокого уровня, напоминает собой объявление класса в парадигме объектно-ориентированного программирования. 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. 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.

D. Некоторые особенности смарт контрактов

Первая особенность, 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.

Следующая особенность, на которую стоит обратить внимание, это разные способы вызова функций другого смарт контракта. Как было показано раннее, 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.

И наконец, вызовы функций других контрактов происходят синхронно, то есть вызывающий код ожидает возвращения из вызываемой функции до того, как перейдет к следующей инструкции. Никакой необходимости в асинхронности нет, поскольку узел выполняющий код некоторого контракта имеет локальный доступ ко всем остальным контрактам, хранящимся в сети.

II. Уязвимости смарт контрактов

В своей работе об атаках на смарт контракты Атзеи, Бартолетти и Кимоли предложили свою систематизацию уязвимостей контрактов, разделив все уязвимости на три уровня: уровень языка Solidity, уровень EVM и уровень блокчейна [2]. Далее в данной работе будут представлены уязвимости, обобщенные именно по такому принципу.

A. Уровень языка Solidity

1) 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:

- ошибка при вызове функции другого контракта посредством ABI,
- пополнение баланса другого контракта порождает вызов его *fallback* функции,
- ошибка в объявлении интерфейса при вызове функции другого контракта посредством API.

2) Повторный вход

Как было сказано ранее, в Ethereum вызовы функций других контрактов происходят синхронно, то есть 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.

3) 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:

- при вызове функции другого контракта напрямую через API исключения наследуются, то есть если в вызываемой функции будет сгенерировано исключение, то выполнение кода вызываемой функции прекратится, и изменения будут откатаны,
- 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.

4) 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, что хватает только на некоторые байт код операции. Таким образом, 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.

5) Хранение секретов

Как было сказано в первой части статьи, переменные смарт контракта могут быть помечены как публичные или приватные. Тем не менее, приватность переменной не гарантирует ее секретность, поскольку состояние контракта хранится в открытом реестре.

B. Уровень EVM

1) 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.

C. *Уровень блокчейна*

1) Непредсказуемость состояния контракта

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 состоянии 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.

2) 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 так же создает возможность использования майнером своего положения в силу некоторой свободы в выставлении временной метки для блока.

III. Атаки на смарт контракты

A. *DAO*

18 июня 2016 года была осуществлена атака на крауд-фандинговый контракт, в результате которой злоумышленник успешно перевел на свой адрес около $60 млн [8]. Рассмотрим упрощенный вариант этого контракта, содержащий те же уязвимости, и соответствующие атаки. Ниже представлен контракт *DAOExample*:

```
5  contract DAOExample {
6      mapping (address => uint) credit;
7
8      function donate(address to) public payable {
9          credit[to] += msg.value;
10     }
11
12     function withdraw(uint amount) public payable {
13         if (credit[msg.sender] >= amount) {
14             msg.sender.call.value(amount)('');
15
16             credit[msg.sender] -= amount;
17         }
18     }
19
20     function queryCredit() public view returns (uint) {
21         return credit[msg.sender];
22     }
23 }
```

Рис. 1. Атака на DAO. DAOExample

Атака №1

Рассмотрим следующий смарт контракт, который злоумышленнику необходимо опубликовать в сети:

```
29 contract MaliciousContract {
30     DAOExample target = DAOExample(0x692a70D2e424a56D2C6C27aA97D1a86395877b3A);
31     address payable owner;
32
33     constructor() public {
34         owner = msg.sender;
35     }
36
37     function() payable external {
38         target.withdraw(target.queryCredit());
39     }
40
41     function finalize() public payable {
42         address(owner).send(address(this).balance);
43     }
44 }
```

Рис. 2. Атака на DAO №1

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. 1 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:

- закончится *gas*,
- баланс *DAOExample* обнулится.

B. *King of the Ether Throne*

King of the Ether Throne [5] 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 на рис. 3.

Заметим, 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. 4).

```
5  contract KotET {
6      address payable public king;
7      uint public claimPrice = 100;
8      address owner;
9
10     constructor() public {
11         owner = msg.sender;
12         king = msg.sender;
13     }
14
15     function() external payable {
16         require(msg.value >= claimPrice, 'Not enough value provided');
17         uint compensation = calculateCompenstation();
18         king.send(compensation);
19         king = msg.sender;
20         calculateNewPrice();
21     }
22
23     function calculateCompenstation() public returns (uint) {...}
26     function calculateNewPrice() public {...}
29 }
```

Рис. 3. Атака на King of the Ether Throne. KotET

```
15     function() external payable {
16         require(msg.value >= claimPrice, 'Not enough value provided');
17         uint compensation = calculateCompenstation();
18         (bool success, ) = address(king).call.value(compensation)('');
19         require(success, 'Failed to send compensation');
20         king = msg.sender;
21         calculateNewPrice();
22     }
```

Рис. 4. KotET. Модификация

Однако и такая реализация имеет недочет. Она уязвима относительно DoS-атаки. Для того чтобы осуществить такую атаку необходимо опубликовать следующий контракт:

```
32 contract MaliciousContract {
33     function unseatKing(address a, uint w) public {
34         a.call.value(w);
35     }
36
37     function() external payable {
38         require(false);
39     }
40 }
```

Рис. 5. Атака на KotET

После того, как злоумышленник станет королем посредством вызова функции unseatKing данного контракта, контракт KotET перестанет принимать новых королей.

C. GovernMental

GovernMental [6] 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. На рис. 6 представлены основные функции упрощенного контракта.

```
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. 6. Governmental

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.
- 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 уязвимости 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.

Подводя итог, можно сказать, что хоть и сама 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 ЛИТЕРАТУРЫ

[1] Luu L. et al. Making smart contracts smarter //Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016. С. 254-269

[2] 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. С. 164-186.

[3] Andrychowicz M. et al. Secure multiparty computations on bitcoin //2014 IEEE Symposium on Security and Privacy. IEEE, 2014. С. 443-458.

[4] Boneh D., Naor M. Timed commitments //Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2000. С. 236-254.

[5] King of the Ether Throne: Post mortem investigation. URL: https://www.kingoftheether.com/postmortem.html

[6] GovernMental. URL: http://governmental.github.io/GovernMental/

[7] Buterin, V.: Ethereum: a next generation smart contract and decentralized applicationplatform (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper

[8] Understanding the DAO attack. URL: https://www.coindesk.com/understanding-dao-hack-journalists