

Транзакции - это чуть ли не самый "главный" объект в сети Bitcoin, да и в других блокчейнах тоже. Поэтому я решил, что если и писать про них целую главу, то тогда нужно рассказать и показать вообще все, что можно. В частности то, как они строятся и работают на уровне протокола.

Ниже я объясню, каким образом формируется транзакция, покажу как она подписывается и продемонстрирую механизм общения между нодами.

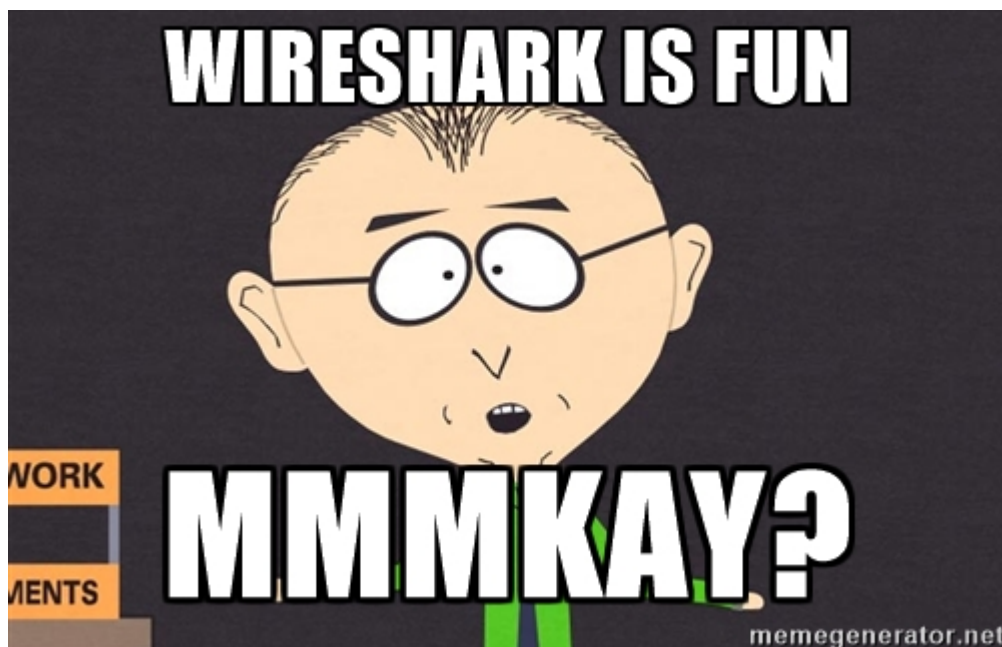


Table of content

1. Keys and address
2. Searching for nodes
3. Version handshake
4. Setting up a connection
5. Making transaction
6. Signing transaction
7. Sniff & spoof
8. Sending transaction
9. Links

Keys and address

Для начала создадим новую пару ключей и адрес. Как это делается я рассказывал в главе [Bitcoin in a nutshell - Cryptography](#), так что здесь все должно быть понятно. Для ускорения процесса возьмем вот этот [набор инструментов для Bitcoin](#), написанный самим [Виталиком Бутериным](#), хотя при желании вы можете воспользоваться уже написанными [фрагментами кода](#).

```

$ git clone https://github.com/vbuterin/pybitcointools
$ cd pybitcointools
$ sudo python setup.py install
$ python
Python 2.7.12 (default, Jul 1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from bitcoin import *
>>> private_key = "28da5896199b85a7d49b0736597dd8c0d0c0293f130bf3e3e1d102e0041b1293"
>>> public_key = privtopub(private_key)
>>> public_key
'0497e922cac2c9065a0cac998c0735d9995ff42fb6641d29300e8c0071277eb5b4e770fcc086f322339bdefef4d5b51a23d88755969d28e965daca5d0d2a0e09'
>>> address = pubtoaddr(public_key)
>>> address
'1LwPhYQi4BRBuuyWSGVeb6kPrTqpSVmoYz'

```

Я [скинул](#) на адрес `1LwPhYQi4BRBuuyWSGVeb6kPrTqpSVmoYz` 0.00012 BTC, так что теперь можно экспериментировать по полной программе.

Searching for nodes

Вообще говоря, это хорошая задача на подумать: *как найти других участников сети при том, что сеть децентрализована?* Подробнее про это можете почитать [здесь](#), скажу заранее, совсем децентрализованного решения пока что не существует.

Я покажу два способа. Первый - это *DNS seeding*. Суть в том, что есть некоторые *доверенные* адреса, такие как:

- bitseed.xf2.org
- dnsseed.bluematt.me
- seed.bitcoin.sipa.be
- dnsseed.bitcoin.dashjr.org
- seed.bitcoinstats.com

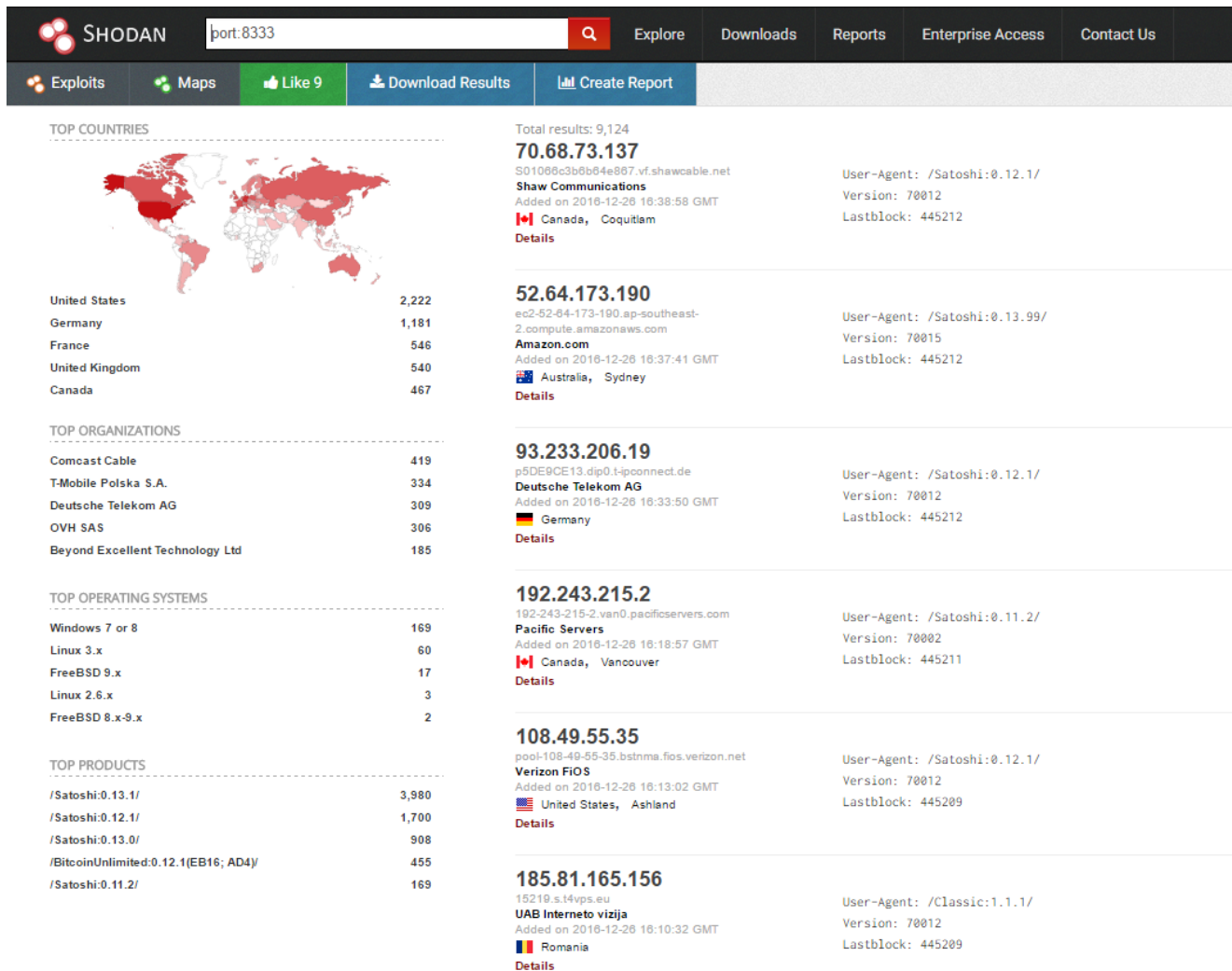
Они захардкожены в [chainparams.cpp](#) и командой `nslookup` можно получить от них адреса нод.

```

$ nslookup bitseed.xf2.org
Non-authoritative answer:
Name:   bitseed.xf2.org
Address: 76.111.96.126
Name:   bitseed.xf2.org
Address: 85.214.90.1
Name:   bitseed.xf2.org
Address: 94.226.111.26
Name:   bitseed.xf2.org
Address: 96.2.103.25
...

```

Другой способ не такой умный и на практике не используется, но в учебных целях он подходит даже лучше. Заходим на [Shodan](#), регистрируемся, авторизуемся и в строке поиска пишем `port:8333`. Это стандартный порт для `bitcoind`, в моем случае нашлось примерно 9.000 нод:



SHODAN port:8333 [Search] Explore Downloads Reports Enterprise Access Contact Us

Exploits Maps Like 9 Download Results Create Report

TOP COUNTRIES

Country	Count
United States	2,222
Germany	1,181
France	546
United Kingdom	540
Canada	467

TOP ORGANIZATIONS

Organization	Count
Comcast Cable	419
T-Mobile Polska S.A.	334
Deutsche Telekom AG	309
OVH SAS	306
Beyond Excellent Technology Ltd	185

TOP OPERATING SYSTEMS

OS	Count
Windows 7 or 8	169
Linux 3.x	60
FreeBSD 9.x	17
Linux 2.6.x	3
FreeBSD 8.x-9.x	2

TOP PRODUCTS

Product	Count
/Satoshi:0.13.1/	3,980
/Satoshi:0.12.1/	1,700
/Satoshi:0.13.0/	908
/BitcoinUnlimited:0.12.1(EB16; AD4)/	455
/Satoshi:0.11.2/	169

Total results: 9,124

70.68.73.137
S01086c3b6b64e867.vf.shawcable.net
Shaw Communications
Added on 2016-12-26 16:38:58 GMT
Canada, Coquitlam
Details
User-Agent: /Satoshi:0.12.1/
Version: 70012
Lastblock: 445212

52.64.173.190
ec2-52-64-173-190.ap-southeast-2.compute.amazonaws.com
Amazon.com
Added on 2016-12-26 16:37:41 GMT
Australia, Sydney
Details
User-Agent: /Satoshi:0.13.99/
Version: 70015
Lastblock: 445212

93.233.206.19
p5DE9CE13.dip0.lipconnect.de
Deutsche Telekom AG
Added on 2016-12-26 16:33:50 GMT
Germany
Details
User-Agent: /Satoshi:0.12.1/
Version: 70012
Lastblock: 445212

192.243.215.2
192-243-215-2.van0.pacificservers.com
Pacific Servers
Added on 2016-12-26 16:18:57 GMT
Canada, Vancouver
Details
User-Agent: /Satoshi:0.11.2/
Version: 70002
Lastblock: 445211

108.49.55.35
pool-108-49-55-35.bstnma.fios.verizon.net
Verizon FIOS
Added on 2016-12-26 16:13:02 GMT
United States, Ashland
Details
User-Agent: /Satoshi:0.12.1/
Version: 70012
Lastblock: 445209

185.81.165.156
15219.s14vps.eu
UAB Interneto vizija
Added on 2016-12-26 16:10:32 GMT
Romania
Details
User-Agent: /Classic:1.1.1/
Version: 70012
Lastblock: 445209

Version handshake

Установка соединения между нодами начинается с обмена двумя сообщениями. Первым отправляется [version message](#), а в качестве ответа на него используется [verack message](#). Вот иллюстрация процесса *version handshake* из [Bitcoin wiki](#):

When the local peer **L** connects to a remote peer **R**, the remote peer will not send any data until it receives a version message.

- **L** -> **R** Send version message with the local peer's version
- **R** -> **L** Send version message back
- **R** Sets version to the minimum of the 2 versions
- **R** -> **L** Send verack message
- **L** Sets version to the minimum of the 2 versions

Это делается в первую очередь для того, чтобы ноды узнали, какой версией протокола пользуется их "собеседник" и могли общаться на одном языке.

Setting up a connection



Каждое сообщение в сети [должно представляться](#) в виде `magic + command + lenght + checksum + payload`, за это отвечает функция `makeMessage`. Этой функцией мы еще воспользуемся, когда будем отправлять транзакцию.

В коде будет постоянно использоваться библиотека [struct](#). Она отвечает за то, чтобы представлять параметры в правильном формате. Например `struct.pack("q", timestamp)` записывает текущее UNIX время в `long long int`, как этого и требует протокол.

```

import time
import socket
import struct
import random
import hashlib

def makeMessage(cmd, payload):
    magic = "F9BEB4D9".decode("hex") # Main network ID
    command = cmd + (12 - len(cmd)) * "\00"
    length = struct.pack("I", len(payload))
    check = hashlib.sha256(hashlib.sha256(payload).digest()).digest()[:4]
    return magic + command + length + check + payload

def versionMessage():
    version = struct.pack("i", 60002)
    services = struct.pack("Q", 0)
    timestamp = struct.pack("q", time.time())

    addr_rcv = struct.pack("Q", 0)
    addr_rcv += struct.pack(">16s", "127.0.0.1")
    addr_rcv += struct.pack(">H", 8333)

    addr_from = struct.pack("Q", 0)
    addr_from += struct.pack(">16s", "127.0.0.1")
    addr_from += struct.pack(">H", 8333)

    nonce = struct.pack("Q", random.getrandbits(64))
    user_agent = struct.pack("B", 0) # Anything
    height = struct.pack("i", 0) # Block number, doesn't matter

    payload = version + services + timestamp + addr_rcv + addr_from + nonce + user_agent + height

    return payload

if __name__ == "__main__":

    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(("93.170.187.9", 8333))

    sock.send(makeMessage("version", versionMessage()))
    sock.recv(1024) # receive version message
    sock.recv(1024) # receive verack message

```

Теперь открываем Wireshark, ставим фильтр `bitcoin` или `tcp.port == 8333` и смотрим на получившиеся пакеты. Если все сделано верно, то, во-первых, будет верно определен протокол, *user-agent*, *block start height* и так далее. Во вторых, как и обещалось, вам прилетит ответ в виде сообщений *version* и *verack*. Теперь, когда соединение установлено, можно начинать работу.

75290	3597.475281	192.168.0.107	93.170.187.9	Bitcoin	163 version
75292	3597.547412	93.170.187.9	192.168.0.107	Bitcoin	184 version
75294	3597.634593	93.170.187.9	192.168.0.107	Bitcoin	430 verack, alert, ping


```

> Frame 75290: 163 bytes on wire (1304 bits), 163 bytes captured (1304 bits) on interface 0
> Ethernet II, [REDACTED] Dst: Tp-LinkT_a5:8c:ac (64:70:02:a5:8c:ac)
> Internet Protocol Version 4, Src: 192.168.0.107, Dst: 93.170.187.9
> Transmission Control Protocol, Src Port: 50137, Dst Port: 8333, Seq: 1, Ack: 1, Len: 109
▼ Bitcoin protocol
  Packet magic: 0xf9beb4d9
  Command name: version
  Payload Length: 85
  Payload checksum: 0xcc444863
  ▼ Version message
    Protocol version: 60002
    > Node services: 0x0000000000000000
    Node timestamp: Dec 30, 2016 17:10:50.000000000 Russia TZ 2 Standard Time
    > Address as receiving node
    > Address of emitting node
    Random nonce: 0x9408a4e1df9c2216
    ▼ User agent
      Count: 0
      String value:
      Block start height: 0

```

Making transaction

Перед созданием транзакции еще раз открываем [спецификацию](#) и внимательно ее придерживаемся. Отклонение на 1 байт уже делает транзакцию невалидной, так что нужно быть предельно аккуратным.

Для начала зададим адреса, приватный ключ и хэш [транзакции](#), на которую мы будем ссылаться:

```

previous_output = "60ee91bc1563e44866c66937b141e9ef4615a272fa9d764b9468c2a673c55e01"
receiver_address = "1C29gpF5MkEPReCiGtKVXWdAmNiQ4PBMH"
my_address = "1LwPhYQi4BRBuuyWSGVeb6kPrTqpSVmoYz"
private_key = "28da5896199b85a7d49b0736597dd8c0d0c0293f130bf3e3e1d102e0041b1293"

```

Далее создадим транзакцию в *raw* виде, то есть пока что неподписанную. Для этого достаточно просто следовать спецификации:


```
def txnMessage(previous_output, receiver_address, my_address, private_key):
    receiver_hashed_pubkey= base58.b58decode_check(receiver_address)[1:].encode("hex")
    my_hashed_pubkey = base58.b58decode_check(my_address)[1:].encode("hex")

    # Transaction stuff
    version = struct.pack("<L", 1)
    lock_time = struct.pack("<L", 0)
    hash_code = struct.pack("<L", 1)

    # Transactions input
    tx_in_count = struct.pack("<B", 1)
    tx_in = {}
    tx_in["outpoint_hash"] = previous_output.decode('hex')[::-1]
    tx_in["outpoint_index"] = struct.pack("<L", 0)
    tx_in["script"] = ("76a914%s88ac" % my_hashed_pubkey).decode("hex")
    tx_in["script_bytes"] = struct.pack("<B", (len(tx_in["script"])))
    tx_in["sequence"] = "ffffffff".decode("hex")

    # Transaction output
    tx_out_count = struct.pack("<B", 1)

    tx_out = {}
    tx_out["value"] = struct.pack("<Q", 1000) # Send 1000 satoshis
    tx_out["pk_script"] = ("76a914%s88ac" % receiver_hashed_pubkey).decode("hex")
    tx_out["pk_script_bytes"] = struct.pack("<B", (len(tx_out["pk_script"])))

    tx_to_sign = (version + tx_in_count + tx_in["outpoint_hash"] + tx_in["outpoint_index"] +
                  tx_in["script_bytes"] + tx_in["script"] + tx_in["sequence"] + tx_out_count +
                  tx_out["value"] + tx_out["pk_script_bytes"] + tx_out["pk_script"] + lock_time +
                  hash_code)
```

Заметьте, что в поле `tx_in["script"]` написано отнюдь не `<Sig> <PubKey>`, как вы, наверное, ожидали. Вместо этого указан **блокирующий скрипт выхода, на который мы ссылаемся**, в нашем случае это `OP_DUP OP_HASH160 dab3cccc50d7ff2d1d2926ec85ca186e61aef105 OP_EQUALVERIFY OP_CHECKSIG`.

BTW нет никакой разницы между привычным `OP_DUP OP_HASH160 dab3cccc50d7ff2d1d2926ec85ca186e61aef105 OP_EQUALVERIFY OP_CHECKSIG` и `76a914dab3cccc50d7ff2d1d2926ec85ca186e61aef105s88ac` - во втором случае просто используется [специальная кодировка](#) для экономии места:

```
0x76 = OP_DUP
0xa9 = OP_HASH160
0x14 = далее следует 14 байт информации
dab3cccc50d7ff2d1d2926ec85ca186e61aef105s88ac
...
```

Signing transaction

Теперь самое время подписать транзакцию, здесь все довольно просто:

```
hashed_raw_tx = hashlib.sha256(hashlib.sha256(tx_to_sign).digest()).digest()
sk = ecdsa.SigningKey.from_string(private_key.decode("hex"), curve = ecdsa.SECP256k1)
vk = sk.verifying_key
public_key = ('\04' + vk.to_string()).encode("hex")
sign = sk.sign_digest(hashed_raw_tx, sigencode=ecdsa.util.sigencode_der)
```

После того, как получена подпись для *raw transaction*, можно заменить unlocking script на настоящий и привести транзакцию к окончательному виду:

```
sigscript = sign + "\01" + struct.pack("<B", len(public_key.decode("hex"))) +
public_key.decode("hex")

real_tx = (version + tx_in_count + tx_in["outpoint_hash"] + tx_in["outpoint_index"] +
struct.pack("<B", (len(sigscript) + 1)) + struct.pack("<B", len(sign) + 1) + sigscript +
tx_in["sequence"] + tx_out_count + tx_out["value"] + tx_out["pk_script_bytes"] +
tx_out["pk_script"] + lock_time)

return real_tx
```

Sniff & spoof

Здесь нужно пояснить одну деталь. Я думаю вы понимаете, зачем мы вообще подписываем транзакции. Это делается для того, чтобы никто не смог изменить наше сообщение и отправить его дальше по сети, потому что изменится подпись сообщения и так далее.

Но если вы внимательно читали, то запомнили, что мы подписываем ненастоящую транзакцию, которая в конечном итоге будет отправлена другим нодам, а ее модификацию, где в unlocking script указан locking script из выхода, на который мы ссылаемся. В принципе понятно, почему это происходит: в настоящий unlocking script должна быть записана эта самая подпись, и получается замкнутый круг: для правильной подписи нужен правильный unlocking script, для правильного unlocking script нужна правильная подпись. Так что Сатоши пошел на компромисс и разрешил пользоваться не совсем "настоящими" подписями.

Поэтому может случиться так, что кто-нибудь в сети поймает наше сообщение, изменит unlocking script и отправит отредактированное сообщение дальше. Никто из нод не сможет этого проверить, потому что подпись не "защищает" unlocking script. Эта уязвимость называется **Transaction malleability**, подробнее про нее вы можете почитать [здесь](#) или посмотреть доклад с Black Hat USA 2014 - [Bitcoin Transaction Malleability Theory in Practice](#).

TL;DR Если вы пользуетесь стандартными скриптами вроде P2PKH, то вам ничего не грозит. В противном случае стоит быть аккуратным.

Sending transaction

Отправка транзакции в сеть производится точно так же, как и в случае с *version message*:


```

if __name__ == "__main__":

    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(("70.68.73.137", 8333))

    sock.send(makeMessage("version", versionMessage()))
    sock.recv(1024) # version
    sock.recv(1024) # verack

    # Transaction options
    previous_output = "60ee91bc1563e44866c66937b141e9ef4615a272fa9d764b9468c2a673c55e01"
    receiver_address = "1C29gpF5MkEPREciGtKVXwWdAmNiQ4PBMH"
    my_address = "1LwPhYQi4BRBuuyWSGVeb6kPrTqpSVmoYz"
    private_key = "28da5896199b85a7d49b0736597dd8c0d0c0293f130bf3e3e1d102e0041b1293"

    txn = txnMessage(previous_output, receiver_address, my_address, private_key)
    print "Signed txn:", txn

    sock.send(makeMessage("tx", txn))
    sock.recv(1024)

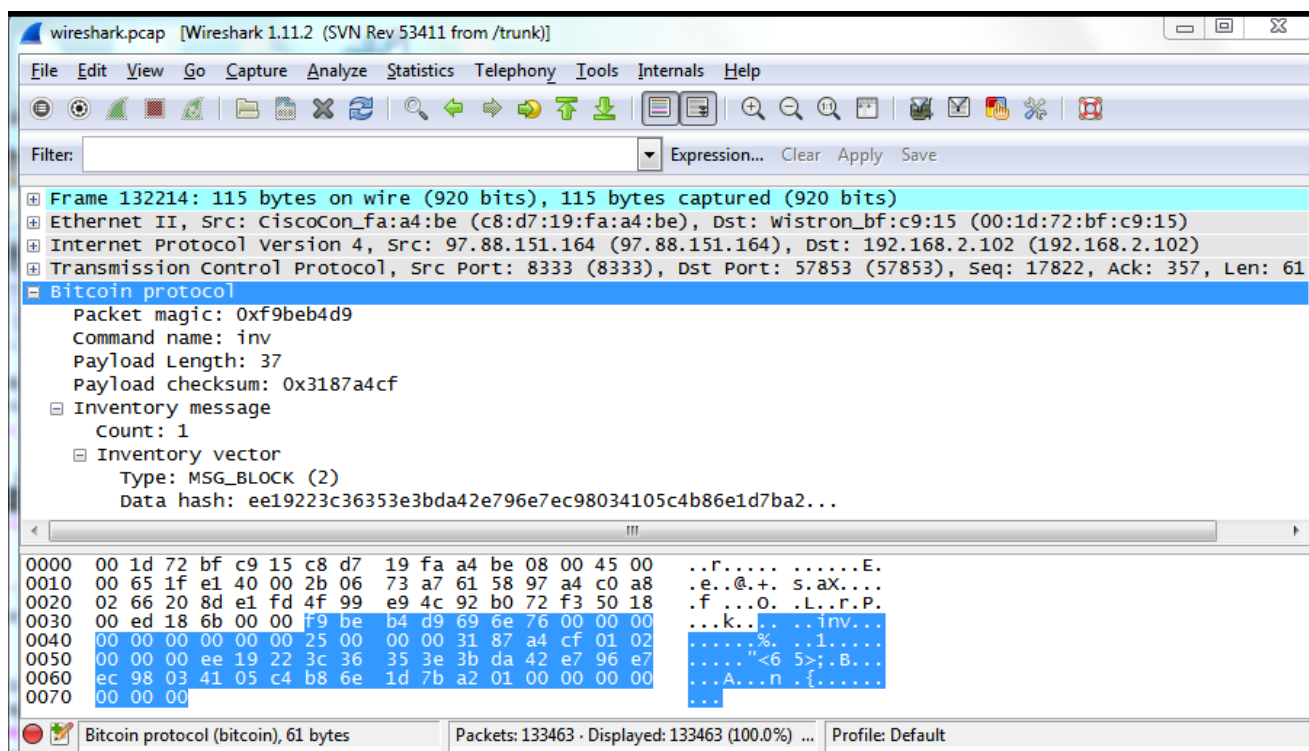
```

Запускаем получившийся код и бежим смотреть на пакеты. Если все сделано верно, то в качестве ответа на ваше сообщение придет [inv message](#) (в противном случае был бы [reject message](#)). Интересный факт - каждая нода, при получении свежей транзакции проверяет ее на валидность (процесс описан в [Bitcoin in a nutshell - Mining](#)), поэтому если вы где-то ошиблись, то вас об этом мгновенно оповестят:

	26	2.101161	192.168.0.107	70.68.73.137	Bitco...	302 tx
	28	3.198843	70.68.73.137	192.168.0.107	Bitco...	115 inv
<div> <div>Bitcoin protocol</div> <div> <div>Packet magic: 0xf9beb4d9</div> <div>Command name: tx</div> <div>Payload Length: 224</div> <div>Payload checksum: 0x69d32a8c</div> </div> <div> <div>Tx message</div> <div> <div>Transaction version: 1</div> <div>Input Count: 1</div> <div>Transaction input</div> <div> <div>Previous output</div> <div> <div>Script Length: 139</div> <div>Signature script: 48304502210086fb3df5d4cc282649817051190536eaa2bb...</div> <div>Sequence: 4294967295</div> </div> <div>Output Count: 1</div> <div>Transaction output</div> <div> <div>Value: 1000</div> <div>Script Length: 25</div> <div>Script: 76a91478e10cf8e4bd38266d8fd4ed5c8b430d30a3cde888...</div> </div> </div> <div>Block lock time or block ID: 0</div> </div> </div> </div>						
0000	64	70	02	a5	8c	ac 00 1d 60 1b 8f be 08 00 45 00 dp.....`.....E.
0010	01	20	67	bc	40	00 80 06 00 00 c0 a8 00 6b 46 44 . g.@... ..kFD
0020	49	89	e7	b6	20	8d 40 b2 f9 f3 86 14 f3 a4 50 18 I... .@.P.
0030	00	ff	51	f3	00	00 f9 be b4 d9 74 78 00 00 00 00 ..Q... ..tx....
0040	00	00	00	00	00	00 e0 00 00 00 69 d3 2a 8c 01 00i.*...
0050	00	00	01	01	5e	c5 73 a6 c2 68 94 4b 76 9d fa 72^..s. .h.Kv..r
0060	a2	15	46	ef	e9	41 b1 37 69 c6 66 48 e4 63 15 bc ..F..A.7 i.fH.c..
0070	91	ee	60	00	00	00 00 8b 48 30 45 02 21 00 86 fb ..`.....H0E.!...
0080	3d	f5	d4	cc	28	26 49 81 70 51 19 05 36 ea a2 bb =...(&I. pQ..6...
0090	20	60	4d	77	9a	88 54 96 4d 35 2b f7 79 77 02 20 `Mw..T. M5+.yw.
00a0	38	7f	5d	e9	0f	42 f3 45 86 17 88 d3 47 bd f6 bb 8.]..B.EG...
00b0	82	7c	02	88	88	f3 72 95 f1 56 f7 bc a1 b2 d6 f7r. .V.....
00c0	01	41	04	97	e9	22 ca c2 c9 06 5a 0c ac 99 8c 07 .A..."..Z.....
00d0	35	d9	99	5f	f4	2f b6 64 1d 29 30 0e 8c 00 71 27 5.._./..d .)0...q'
00e0	7e	b5	b4	e7	70	fc c0 86 f3 22 33 9b de fe f4 d5 ~...p... ."3.....
00f0	b5	1a	23	d8	87	55 96 9d 28 e9 65 da ca aa 5d 0d ..#..U.. (.e...].

Уже через несколько секунд после отправления транзакции в сеть, ее можно будет [отследить](#), правда сначала она будет числиться неподтвержденной. Потом, спустя какое-то время (вплоть до нескольких часов), транзакция будет включена в блок.

Если вы к тому времени не закроете Wireshark плюс в сообщении *version* укажете текущую высоту блокчейна, то вам придет уведомление о новом блоке в виде все того же *inv message*, но на этот раз с `TYPE = MSG_BLOCK` (я его закрыл, поэтому ниже скриншот из блога [Ken Shirriff](#)):



В **Data hash** вы можете видеть длинную строку, которая на самом деле является заголовком нового блока в *little endian* форме. В данном случае это блок [#279068](#) с заголовком `0000000000000001a27b1d6eb8c405410398ece796e742da3b3e35363c2219ee`. Куча ведущих нулей - не случайность, а результат майнинга, о котором я расскажу отдельно.

Но перед этим вам нужно разобраться с самим блокчейном, блоками, их заголовками и так далее. Поэтому следующая глава: [Bitcoin in a nutshell - Blockchain](#)

Links

- [Bitcoins the hard way: Using the raw Bitcoin protocol](#)
- [Analyzing Bitcoin Network Traffic Using Wireshark](#)
- [How the Bitcoin protocol actually works](#)
- [How does a Bitcoin node find its peers?](#)
- [Bitcoin Developer Reference. P2P network](#)
- [Redeeming a raw transaction step by step](#)