Randpay

Large-scale micropayment subsystem based on the Emercoin cryptocurrency

This article describes a system for aggregating payments based on a zero-trust lottery protocol. We demonstrate its applicability to very large-scale settlements amounting to trillions of transactions per year. Under the protocol, the payment recipient creates a unique payment address and defines a space of possible payment addresses including this one. The payer chooses a random address from the specified space, creates a payment transaction to the selected address, and then sends the transaction to the payment recipient. If the address specified in the received transaction matches the previously created one, the recipient signs the transaction and publishes it to the blockchain, and payment is transferred. If the addresses do not match, the transaction is deemed invalid and is ignored. In a typical Randpay scenario, the payment is unlikely in each particular "lottery" instance. But as the number of such instances is high, the balances of all participants will be updated in a fair way. This article presents a mathematical model proving that, for all the participants, Randpay will statistically converge to the fair amounts they would have received or sent without using this subsystem. In addition, Randpay reduces payers' transactional costs, making it economically viable, especially for micropayments.

The problem of scaling

Blockchain-based decentralized payment systems are becoming increasingly popular. Optimists and crypto-enthusiasts even argue that all banking systems will soon become obsolete, as businesses will be using Bitcoin or other cryptocurrencies for direct settlements. But blockchain systems are initially limited in scalability due to their architecture, as the blockchain must contain all transactions that have ever taken place. Using the telecom industry as an example, you can easily calculate that if every phone call ends in a settlement, then, with a transaction size of 200 bytes — usually more — and 2 trillion calls a year across the global telecom industry, the blockchain will grow by 400 terabytes per year. This is obviously unacceptable. Mechanisms to reduce transaction size do exist. Segwit, for example, reduces the transaction size by almost half. But even 200 terabytes per year instead of 400 still constitutes an insurmountable barrier to using cryptocurrencies in the telecom industry — if every call is paid — and in other areas that require large-scale payments.

Even today, in an age when cryptocurrency payments are far from mainstream, major cryptocurrency blockchains already take up hundreds of gigabytes, and maintaining a full cryptocurrency node is becoming quite expensive. Imagine what will happen when cryptocurrency payments start occurring on a large scale.

In addition, each node of such a distributed system receives all the transactions of all network participants in real time. With a large flow of transactions, this may cause network bandwidth issues.

We can't say that this problem has been overlooked by cryptocurrency developers. From time to time, we see claims that some cryptocurrency is going to have "a bandwidth comparable to that of VISA" or ideas about sharding, i.e. dividing the blockchain database among the participants. But we're yet to see anything that actually works, even as a prototype. In any case, we wish these developers success in making these claims a reality.

As for us, here we will focus on what already exists, works, and can be applied here and now.

We propose a realistic system that solves the problems of scalability via aggregation, i.e. replacing a large number of micropayments with a small number of final ones, which will be included in the blockchain. In addition, with aggregation, one only has to pay transaction fees on the aggregated transactions. In other words, aggregation not only makes the payment system scalable, but also reduces transaction costs per payment.

Lightning Network

The Lightning Network payment aggregation system constitutes a separate network of agents connected via payment channels. Two agents can create a channel by linking their coins into an "open channel transaction". After making payments exclusively between each other, they close the channel by sending the "close channel transaction" to the blockchain. Thus, only two blockchain transactions (open channel and close channel) are created over the whole channel life cycle, and payments take place directly between the agents, without affecting the blockchain network. Such a system has its pros and cons:

Advantages

- Fast transactions — no need to wait for blockchain confirmations.
- Absolute accuracy - the recipient receives the exact amount intended for them in each instance of payment.

Disadvantages

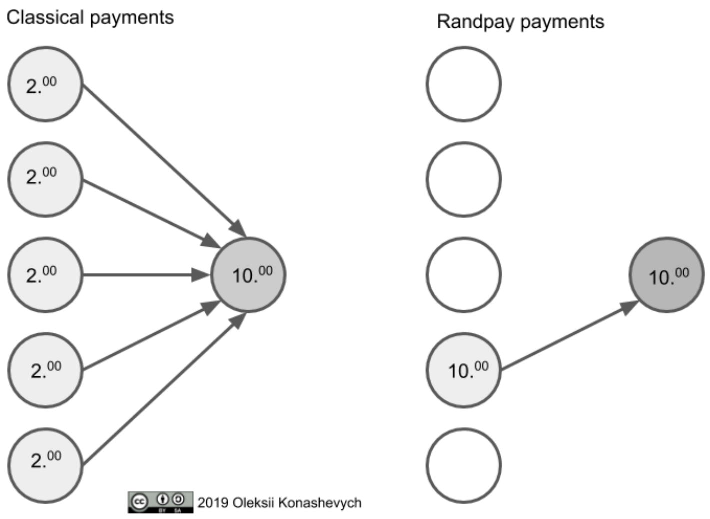
- You need to create and maintain the processing infrastructure - a network of channel operators who are always online.
- Channel operators may ultimately want to get paid for money transfer services within their channels, just like VISA/Mastercard.
- Payments are possible only within the channel network - you can't send a payment "to any address", as is the case with cryptocurrencies today.
- If the channel operator shuts down or intentionally denies service, paying becomes impossible.
- When a channel is created, you have to commit some amount of coins to it. If the counterparty is not cooperating or denies service, this amount may be blocked for a long time — possibly years.
- Denial of service is possible if there is no money in the channel. This can happen during bargain sales or in the case of significant hype around a new product that many people want to buy quickly.
- Lightning is a state-driven protocol: Once you create a channel, you must remember its state somewhere, and you can only forget about it after it is closed. If both parties lose the state, the money bound in the channel may be lost.

Thus, Lightning Network is similar in purpose and structure to payment processing systems such as VISA/Mastercard, but it binds ordinary cryptocurrency holders instead of banks.

Randpay

The Randpay payment aggregation system uses a probabilistic approach based on the lottery payments idea proposed by Ron Rivest, the creator of RC4, RSA, and other cryptosystems.

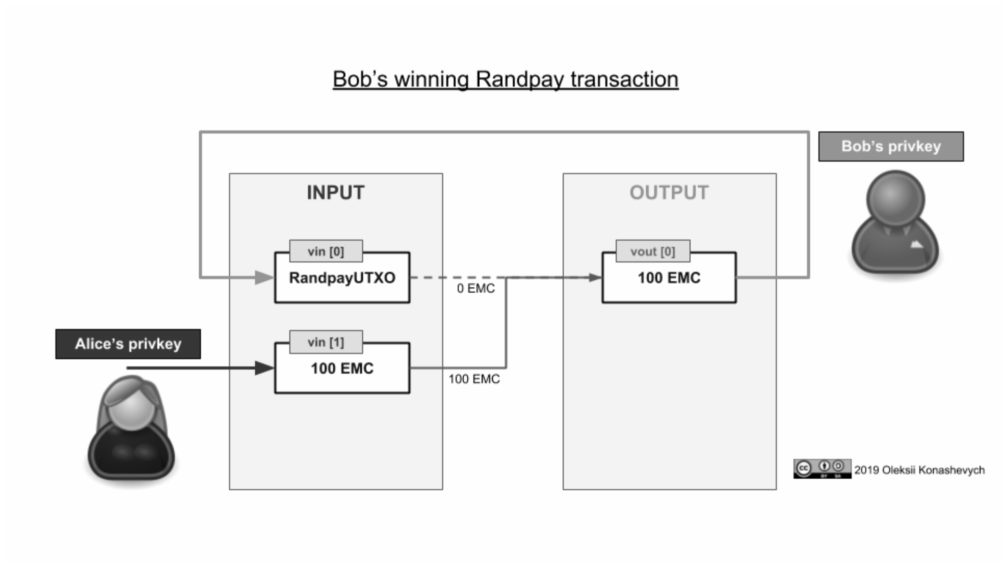
The Idea



The idea is that instead of recording a payment transaction on the blockchain with 100% probability, we issue a “lottery ticket”, which is only recorded when the payer “loses” and the payee “wins” the lottery. In the latter case, only the winning lottery tickets are sent to the blockchain as transactions. “Blank” tickets will not be recorded on the blockchain and may be discarded. At the same time, the expectation of the transferred amount for each lottery payment action is equal to the amount that would have been sent by a regular payment transaction. Using this approach, the number of payments (for the winning tickets) will be quite large, but the payments themselves will be recorded in the blockchain much less frequently.

Example

Let's say you have to pay \$0.01 per minute for a telecom service. But transaction costs and cryptocurrency limitations make such payments unprofitable or, in the case of large amounts, impossible. Using the lottery approach, the customer (payer) pays a much higher amount — say, \$100, — but with a probability of 1/10000. Thus, one payment act averages \$0.01 per minute.



In other words, the customer (payer) plays Russian roulette with a revolver that has 9999 empty slots and one loaded with a payment. In a payment act, if no payment is triggered (which happens 9999 of 10000 times), the client "wins" and receives a minute of service for free. If the payment is triggered (with a probability of 1/10000), the server (service/product provider) wins, and a $100 transaction is recorded on the blockchain.

Thus, there will be situations in which the customer receives the session completely free of charge and others when he or she massively overpays for one minute. But on average, if the customer uses services — not necessarily from the same supplier — on a regular basis, the amount of the payments approaches the fair value, i.e. one that would be the case for actual per-minute payments. At the same time, the number of actual transactions decreases by 99.99% (in this example), proportionally reducing transaction costs.

Thus, using Randpay in a telecom system with the above parameters will only grow the blockchain by 40 GB for the 2 trillion yearly calls. This amount isn't small, but is quite acceptable for practical uses. Increasing the aggregation factor (e.g., when the per-minute rate drops below $0.01) can reduce blockchain inflation even further. This can be combined with other ways of combating blockchain inflation, such as Segwit and transaction optimization.

Error

Obviously, due to the randomness of the Randpay payment process, the actual amount collected by the payment recipient will equal the sum of the winning lottery tickets. That sum will be different from the fair amount that would have been received if actual payments had been made per minute. But as the service is provided, the amount received will grow linearly, while the deviation will grow as a square root. In other words, as the service is actively used, the relative error will approach zero. If needed, you can then enter this inaccuracy in your accounting documents under "unexpected operating expenses" or simply specify the real amount.

Let's consider the advantages and disadvantages of the Randpay system compared to Lightning Network:

Advantages

- No need to create and maintain a separate network of channel operators. You only need a regular Emer wallet to use Randpay.
- Hence no need to pay such network operators.
- This means no risk factors associated with such operators.
- Unlimited peer-to-peer transactions, just as originally planned for cryptocurrencies.
- The ultimate (average) payment amount for each action may be less than the cryptocurrency's smallest unit (Satoshi).
- No need to tie up money in channels for months or years. Whether the ticket has "won" or not usually becomes clear within a minute and, if not, the payer can reuse the money for another ticket.
- Denial of service is not possible, even if there is no money in the channel.
- Unlike Lightning Network, which creates two transactions — for opening and closing the channel respectively, — Randpay creates one and is thus twice as efficient at the same aggregation factor.
- Randpay is a stateless protocol. In other words, you don't need to set up a channel, enter into any kind of financial arrangement with the counterparty, remember the channel's state, or close it in a specific manner. It's a case of "pay and forget about it".
- Nor do you need to complete the payment separately. This is especially helpful if you have an unreliable network connection. Either party can disconnect at any time, and this won't affect the payment.

Disadvantages

- To make a payment secure, you need to wait for transaction confirmations via block closing, as is the case with regular cryptocurrency payments. It is possible to make it work without confirmations, but this will require additional fraud protection measures. See "Security" below.
- The actual payment amount will not be equal to the "fair" one, although it will approach it as the service is continually used.

The idea behind the protocol and a naive algorithm

Although the Randpay protocol is original in design, it stems from the ideas behind two known cryptographic protocols:

- Coin flipping by telephone esoteric protocol.
- CHAP authentication protocol.

In the Randpay protocol, the server (payee) gives the client a "challenge", and the client must provide a solution. The protocol does not rely on any trust between the client and the server, and the challenge is to pick an address. The protocol uses the following algorithm:

1. The payee creates a new P2PKH payment address and a corresponding private key.
2. The payee then generates a lottery ticket request (addrchap address challenge), i.e. a space of possible addresses, one of which is the address [1] for which they have the private key. This is a

winning address for the payee and a losing one for the payer. The payee sends the request directly to the payer, i.e. without involving the cryptocurrency or the blockchain.

3. The payer generates a random payment address within the specified space, creates a payment transaction to this address — the lottery ticket, — and sends the transaction privately in response to the payee.
4. The payee validates the transaction and, if it is correct, provides the product or service to the customer.
5. If the payer does not guess the address, i.e. makes the payment to an address for which the payee does not have a private key, the payee cannot get the money and is forced to reject the payment transaction, i.e. the payer gets the product or service for free. If the payer does guess the address, the recipient publishes the transaction in the blockchain, thus collecting the payer's money.

Interestingly, in Randpay the payer benefits by choosing an address that the payee does NOT own (i.e. guessing wrong), while in classic CHAP schemes the customer tries to give the right answer (i.e. to guess correctly).

Mathematically, the payee will benefit the most by distributing the winning addresses uniformly in the address space, and the payer by choosing a uniformly random address from the proposed space. In this case, the mean payment amount is equal to the mathematical expectation, i.e. fair. If either party tries to manipulate the distribution in any way, it allows the other party to come up with a strategy that exploits this manipulation in their own favor. For example, if the payee always creates even addresses at step 1, the payer can find out (or assume) this and start always picking odd addresses in step 3, thus never having to pay. The same is true if the payer tries to "optimize" their guessing somehow. In this case, the payee can come up with a special puzzle to make the payer guess right as often as possible. Thus, the Nash equilibrium is achieved when both parties follow the uniform distribution specified in the protocol, and if any of them tries to manipulate the distribution, it is not only useless, but potentially damaging for the manipulator.

Other fraud tactics, such as reducing transaction amounts, not signing the transaction, UTXO double-spending, or using an address outside the specified space, can be easily detected, and the payee can discontinue servicing such a fraudulent customer.

The above naive algorithm does work, but it has a serious drawback that makes it hardly applicable in practice. The drawback is that the payee can behave uncooperatively by always sending the transaction to the blockchain at step 5, whether the payer has guessed the address or not. Note that the payee will not be able to use the money from the transaction, because neither they nor anyone else has the private key required to spend it. In this case, the money is lost forever. In other words, a non-cooperative payee cannot take the money for themselves but can hurt the payer by destroying their money, even if it is of no use to them.

The real algorithm

The real algorithm is generally similar to the naive one but contains a mechanism that prevents the payee from behaving in the non-cooperative way described above. The idea is to make the payee prove they possess the private key for the payment address, i.e. that the payer has guessed the actual address from the challenge. Here's how it works:

The blockchain's UTXO set has a special fictitious output number 0 from a non-existent transaction with TXID=ECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECECEC ECECECECECECECECECEC.

This special output, henceforth called RandpayUTXO, has the following properties:

- It can be spent multiple times, i.e. it is not marked as spent after spending.
- You can only spend 0 coins from it, i.e. nothing.
- This UTXO can only be present in a transaction's inputs once. It is henceforth called "randpay-in".
- This randpay-in is signed with the private key from the vout[0] address of the current transaction. In other words, the actual payment address for RandpayUTXO's scriptPubKey is taken from the vout[0] script of the current transaction when checking and generating the signature.

When the payer creates a Randpay transaction at step 3, they add one more expenditure from the fictitious RandpayUTXO to the ***vin*** input array, i.e. they add the randpay-in input. By doing so, the payer signs all the transaction inputs, except for this one. It'll be signed by the payee afterwards, if the payee is able to. The payer places the payment address to be guessed in vout[0] of the to-be transaction.

If the payee has received a winning transaction (the payer has guessed the address), they have to sign their randpay-in with the private key of the address to which they were going to receive the money, i.e. the address specified in vout[0], before sending the transaction to the blockchain in step 5. The payee thus proves that they own the private key of the vout[0] address and have the right to receive the payment, i.e. to publish the transaction on the blockchain.

If the payer has not guessed the address, the payee cannot sign randpay-in because they do not have the private key for vout[0]. The blockchain simply won't accept an unsigned transaction. As a result, the payee simply has to ignore the transaction and cannot harm the payer.

Mathematics

Unlike Lightning and other payment systems, Randpay has an additional degree of freedom — the aggregation coefficient. It determines how many payments are convolved/combined into a real payment transaction sent to the blockchain. In the example above, the aggregation coefficient is 10,000. In other words, 10,000 payments of an average of $0.01 each are convolved into one transaction of $100 (let's call this amount *amount*). Thus, it is also the number of times that the actual transaction amount grows. We will refer to this aggregation coefficient as ***risk*** because it multiplies the payer's risk in a Randpay payment act. Thus, in our example, the payer risks an amount of $100, which is to be taken from them with a probability that is inverse to the risk:

$$p = \frac{1}{risk}$$

Accordingly, the expectation of the amount of a single payment is:

$$E = p \cdot amount$$

Let's simplify the model by assuming that all transactions in the series have the parameters *(p, amount)*. We can do this because Randpay payments are mutually independent and can be grouped according to these parameters, after which we can apply the following to each group separately.

Since the Randpay payment sequence follows a binomial distribution, a statistically significant number of payments **n** in the series can be approximated by a normal distribution

$$Bin(n,p) \approx N(np, npq)$$

with expectation

$$E = np$$

and variance

$$\sigma^2 = npq$$

where $q = 1 - p$

Then the standard deviation (error) of the amount actually paid/received, compared to the fair amount, is

$$\sigma = \sqrt{npq}$$

and the relative error (the ratio of the deviation to the amount paid over n payments) is

$$E_{error} = \frac{\sigma}{E} = \frac{\sqrt{npq}}{np} = \sqrt{\frac{npq}{n^2p^2}} = \sqrt{\frac{q}{np}}$$

It is easy to see that the relative error approaches zero as ***n*** approaches infinity:

$$\lim_{n\to\infty} \sqrt{\frac{q}{np}} = 0$$

In other words, the longer Randpay is in use, the closer the average payment amount is to the value that would have been the case with classic payments, where each transaction goes to the blockchain. In Randpay, it is far from each transaction that goes to the blockchain, so it grows ***risk*** times slower, reducing actual transaction costs accordingly:

$$blockchain\_inflation = \frac{tx\_size}{risk}$$

$$avg_{fee} = \frac{fee}{risk}$$

It would seem that by increasing ***risk*** we can easily increase the system's efficiency almost indefinitely, as both blockchain inflation and transaction costs are reduced as the risk increases. But increasing ***risk*** automatically leads to an increase in the ***amount*** and a decrease in the number of transactions. This could be a problem:

- The payer might simply not have enough money to create a lottery payment transaction.
- Increasing the amount encourages the payer to cancel the payment by fraudulent methods, for example by double-spending.
- It may be uncomfortable for the payer to overpay that much, even though the probability is low. No one wants to pay $10,000 for a service worth $0.01.
- On the other hand, the system must undergo several real payments during a reporting period: If an excessively high ***risk*** causes the seller to receive an average of one payment in 10 years, they are likely to go broke before it happens.

In other words, the system limits risk increase by the ***amount*** the payer is ready to risk. Interestingly, the system does not limit its downward movement, i.e. it allows increasing the ***risk*** parameter for nano-payments almost without limit. For example, you can imagine a business that sells information about current stock quotes on a piece-by-piece basis, say at a price of $0.00000001 per quote. Or AI answers to some questions. Or news articles, database samples, etc. This enables profitable business models that would not be feasible in other payment systems due to high transaction costs compared to the amounts paid.

Agent interaction protocol

The general protocol considered here is a recommendation for payer–payee interactions. The specific implementation is up to the developers of systems using this or a similar protocol. We emphasize that these recommendations apply to interactions between agents and not between an agent and an Emer node. The latter will be discussed below in the Randpay API specification.

1. **WANTPAY(double amount, uint32_t risk)** [This step is optional but highly recommended]
   The client (payer) sends this request to the server (payee) to inform that they want to make a Randpay payment for the amount of ***amount***, with the trigger probability of ***p = 1 / risk***. Thus, the actual average amount of a single payment is ***amount / risk***. For example, WANTPAY(100, 100000) means that the client is willing to pay 100 EMC with a probability of 1/100,000, so that the average payment will be 0.001 EMC. The client determines the amount and probability based on the money available in their wallet and their willingness to bear the corresponding risk of a large lump-sum payment. On average, it is beneficial for the client to increase the risk parameter, as it reduces the transaction fee percentage. On the other hand, they are limited by the amount of money that will be blocked for the duration of the "lottery" and the risk of paying this amount.

2. **NEEDPAY(double amount, uint32_t risk, char [] addrchap)**
   By sending this response to WANTPAY, the server informs that it wants to receive a payment of ***amount*** at a probability of ***p = 1 / risk***, for which purpose it offers the client to guess the address from the space of addresses based on the (risk, addrchap) pair. The (amount, risk) parameters the server passes to the client may differ from those the client has offered in WANTPAY. This might happen if the server decides that the amount is too large and the probability too low (risk too high), or if it doesn't like the averaged payment amount, or for some other reason.

3. **PAYMENT(uint32_t risk, char[] rawtx)**
   The client sends the server a Randpay transaction in line with the NEEDPAY requirements. The server sends it to the wallet for verification, and, if the wallet approves (no fraud detected, sufficient amount), it provides the product or service to the client. If the client has guessed the address (at a probability of ***p = 1 / risk***), the wallet automatically signs the transaction and publishes it on the Emercoin blockchain.

Randpay API

This API is an extension of Emercoin's JSON/HTTP API and enables the generation and processing of Randpay transactions by Emer nodes. Applications can use this API to interact with an Emer node to make or receive payments. If the API methods work as intended, they will return the values specified below. In case of an incorrect input, fraud attempts, or runtime errors, it returns a standard error code with an accompanying text message.

randpay_createaddrchap (uint32_t risk, int timeout)

This Emer node method, called by the payee, creates a ***risk***-sized space of possible addresses for the payer. The result is used for the NEEDPAY protocol step.

It creates a private/public key pair for the specified ***risk*** and forms the ***addrchap*** space of "raw" addresses for the payer based on the public key:

$$addrchap = \frac{hash160(Pubkey)}{risk}$$

It doesn't make any changes to the wallet (wallet.dat) or the blockchain, but keeps the following pair:

addrchap -> Privkey

in the RAM for ***timeout*** seconds. This pair will be useful for the payee when receiving a Randpay payment within a randpay_submittx call.

Returned value:

- addrchap — 160 bits in hex encoding

The payee passes the resulting addrchap to the payer during the NEEDPAY step.

randpay_createtx (double amount, char[] addrchap, uint32_t risk, int timeout, bool naive=false)

The payer calls this method to create a transaction (lottery ticket) for the amount of ***amount*** to a payee address generated randomly from ***addrchap***. The inputs used in the transaction are blocked for ***timeout*** (usually around 30–60) seconds. Blocking means that these UTXOs will not be usable in other payments from the same wallet for ***timeout*** seconds. The server must make the payment and take the money during this time, or the money will become available for other payments. The optional naive parameter enables a naive Randpay transaction as described earlier. In this case, the transaction is one input shorter. A naive Randpay transaction is useful for debugging or if the payer is confident in the payee's cooperative behavior.

Algorithm:

- Convert addrchap from hex to binary.
- Generate raw address from (risk, addrchap):
  *uint160 pay_addr = risk * addrchap + GetRand(risk);*
- Create a payment transaction to rand_addr, making sure to store a payment to the above address in vout[0].
- Block the inputs used.
- Sign all transaction inputs.
- If it isn't a naive transaction, create a ***randpay-in*** input by adding an expenditure to vin[0] from RandPayUTXO: EC...EC:0

Returned value:

- Randpay_tx_hex — hex raw transaction

The result of this method is a Randpay payment transaction (lottery ticket) that the payer sends to the recipient during the PAYMENT step.

randpay_submittx(char[] Randpay_tx_hex, uint32_t risk)

Here, the payee's node first verifies the Randpay transaction and then checks whether the payer has guessed the address by comparing it with the previously stored addrchap -> Privkey pair. If the payer was unlucky and the address is the same, then:

- Transfers Privkey to the wallet, thus allowing the payee to spend the money received.
- Signs ***randpay-in*** (for a non-naive transaction) and sends the transaction to the network.

The method supports a special value of ***risk = 0***. This is a special case for light client transactions. It tells the Emer node to validate the transaction and, if ***randpay-in*** is signed, send it to the blockchain. In doing so, the node does not use addrchap -> Privkey pairs, does not sign anything, and does not transfer any Privkey to its wallet. It is assumed that the light wallet will do these things itself.

Returned values:

- amount – the amount from vout[0]
- won — a boolean flag indicating whether the lottery ticket has won

Randpay URI interface

In addition to the above API, an Emercoin wallet contains a mechanism for sending Randpay payments by being called via a URI by an external application (such as a web browser). This is similar to Bitcoin's BIP21 mechanism, also supported by Emercoin. It allows websites to request Emercoin micro-payments via the user's browser by creating a certain URI on the webpage. Once the user clicks on the link, the user's wallet immediately makes a Randpay payment to the website (after a confirmation, of course).

Websites can use this interface to sell access to articles or content (videos, music, etc.). For example, a website may request 0.001 EMC for accessing an article by creating a Randpay request with the parameters of amount = 10 & risk = 10,000. The following is an example of a URI for such a request:

```
emercoin:randpay?amount=10.0&chap=00deadbeef&risk=10000&submit=http%3A%2F%2Frandpay.news.com%3Fid=777%26a
rticle=666
```

Let's examine the Randpay URI structure in detail. Here is what it looks like:

```
emercoin:[//]randpay?amount=DOUBLE&chap=chap_hex&risk=INT[&timeout=INT]&submit=CALLBACK_URI
```

Square brackets indicate optional URI elements. This interface's URI (**amount, chap, risk, timeout**) parameters correspond to a randpay_createtx() call.

- **amount** — payment transaction amount. The payer will pay this if the ticket wins.
- **chap** — lottery ticket requirement generated by the payee (website), in hex encoding.
- **risk** — inverse winning probability.
- **timeout** — number of seconds for which the UTXO outputs participating in the transaction will be blocked.

The **submit** parameter specifies the callback URI. This is where the user's wallet will send the generated Randpay transaction (lottery ticket) via HTTP POST, in hex encoding. Special characters reserved in RFC3986 (e.g. "/:&") must be percent-encoded in order to avoid any conflict between the original and callback URIs.

The wallet user can specify parameters controlling the interface behavior in **emercoin.conf** (default values are given after the "=" sign below):

- **rp_max_amount**=0 — maximum transaction amount that can be sent without confirmation, in EMC.
- **rp_max_payment**=0 — maximum payment that can be sent without confirmation, in EMC.
- **rp_timeout**=30 — number of seconds for blocking inputs used in Randpay transactions.
- **rp_submit**=false — whether the Submit button must be selected automatically in the payment confirmation window. The default choice is CANCEL, i.e., cancel the payment.

Security

Below we consider some possible fraud scenarios on the payer's part and ways to combat them. We also give recommendations allowing the payee to minimize possible fraud losses.

There are two ways of attacking the Randpay subsystem: forming invalid transactions in order to pay less or nothing at all, or attacking the blockchain consensus by canceling the transaction via double spend (DS).

Attacking the transaction:

The attack scenarios discussed here are useless in Randpay because the current code has mechanisms to prevent such attacks. We nevertheless consider them here for educational purposes.

1. Attempt to form a transaction with the ***risk*** value different from the one specified in the transaction details.
   Causes incorrect addrchap extraction in *submittx*, resulting in an error code. The probability that the unpacked addrcahp will match someone else's is negligibly low and can be ignored. Even if that happens, the payee will get the money, which is also a good thing.

2. Attempt to pay less than requested.
   Detected when analyzing ***amount*** returned by *submittx*.
3. Attempt to pay for a transaction with non-existent or previously spent coins.
   Detected during validation in *submittx*. Returns an error code.
4. Attempt to provide addresses belonging to the payee as inputs and steal the payee's money through change.
   Detected during validation in *submittx*. Returns an error code.
5. Attempt to reuse someone else's signature, taken from a previously published transaction, for RandpayUTXO.
   Makes no sense, as the transaction and destination address are created by the payer, so there is simply no point in their using a randpay-in with a signature found somewhere else. They may as well create a naive transaction without any randpay-in at all and spare the payee from proving possession of the address.

Consensus attacks:

Randpay transactions are just as vulnerable to double-spend (DS) consensus attacks as any other payment transaction. In that sense, it’s no better or worse than the latter. The most effective way to fight a consensus attack is to wait for enough blockchain confirmations, just as is the case with any blockchain system. In some cases, though (e.g. when selling call minutes), you have no time to wait for confirmations and have no choice but to work without them. In such situations, Randpay behaves slightly better than the classical mechanism, as it is the payee who decides whether there was a win and whether to publish the transaction on the blockchain, i.e. they have a kind of head start. Below we will consider several DS consensus attack scenarios on non-confirmed Randpay transactions.

1. The payer generates a DS, sends a Randpay payment, and then immediately sends the DS to the blockchain. The result is a logical race between the payee's receiving the DS and completing *submittx*. If the payee is the first to receive the DS, they can track the fraud. Given that the check is done at all times, regardless of whether it is a winning ticket, fraudulent behavior will be detected before the first real payment is made. The payer thus may be able to deceive the payee for the actual amount of payment, i.e. ***amount / risk***.
   Note that the earlier the scammer launches the DS, the more likely it is that the attack will succeed (the DS will end up on the blockchain) but also the more likely it is that the payee will detect the fraud.
   But given that a Randpay transaction does NOT work in most cases, the fraud will most likely be detected on a losing ticket. However, if the network topology is unknown, the payee is on average exposed to the risk of losing half of the actual payment amount, i.e.
   amount / risk / 2.
   To combat this kind of fraud, the payee's strategy is to wait for a random time interval of about 0-10 seconds before providing the product or service, while monitoring the blockchain for a fraudulent DS transaction. If you see one, do not provide the product regardless of which transaction wins the logical race.

2. The payer generates a DS but only sends it to the blockchain when the Randpay transaction they paid with comes to their mempool. In other words, they only attack when they see that the Randpay recipient has won their money, thus trying to get it back. In this situation, the payer

has little chance of getting it back, as they receive their copy of the winning transaction, which has already been distributed over the blockchain, which is likely to reject the DS.
And since Emercoin has a strict transaction priority according to the time of receipt, and since there are no fee-dependent priorities, the scammer cannot raise the priority of their DS transaction by "squeezing" the original transaction out of the mempool.
To minimize this possibility, the payee server's ***connect*** setting has to only allow connections with a few known reliable peers so as to avoid the direct delivery of a winning Randpay transaction to the scammer and to give it time to "spread" over the blockchain instead: The later the cheater gets a Randpay copy via the blockchain, the less chance their DS has to win the race.
Delaying the transfer of the product while holding the amount is also useful when the payee wins.
For new, unknown payers it makes sense to limit ***risk*** so that they do not inflate the win amount (note that if the attack is successful, it wins the full ***amount*** and not ***amount / risk***).

3. PoS minting scam
   In this case the scammer finds a kernel transaction resolving the block and immediately attaches a DS-containing block to it. But instead of sending the block to the blockchain, they make a Randpay purchase. After receiving the product, the scammer publishes the block on the blockchain, thus taking the money back, because the DS transaction will have one confirmation while the original Randpay transaction will have none! But we note that as long as the payer keeps the block from being published, there is a chance that someone else will close it, and the payer will lose the minter's reward for solving it. There are two choices here, similar to those described above:
   - The scammer always publishes the delayed block, thus revealing themselves to the payee, even if Randpay wasn't triggered.
   - The scammer only publishes the delayed block after the transaction appears on the blockchain, thus deleting it from the mempool.

   In both cases, the minter's reward for the block is at risk of not being received due to the delay, as someone else may close the block and receive the reward. The average risk is 1% of the amount for every 6 seconds of delay. And this is the case for every block, not just for those where the payee wins. So the amount of losses must be multiplied by ***risk***:
   loss = risk * block_rewrd * time_delay / 600s.
   Note also that, in order to perform such an attack, scammer has to have a lot of coins that have not been used for a long time, so an attack is by no means always possible. All in all, it's not suitable for systematic theft.

Recommendations to the seller (payee):

The main recommendation is to wait for several confirmations (at least one), just as is the case with ordinary transactions. It'll keep you safe from scammers without causing too much trouble.

If you do need to sell in real time, without waiting for a confirmation:

- Maintain a list of clients (at least by IP addresses) and do not let unverified clients use a high ***risk*** value.
- For unverified clients, delay the delivery of the product or service.
- Track the DS via another recipient wallet that is not related to the primary one.

- Don't let just anyone connect to your main wallet - only keep the ***connect*** setting for peers you tend to trust (e.g. those in the **seed.emercoin.com** pool)
- Finally, if the product or service is not of significant valuable to you and you can afford losing it without compensation, simply accept the risk of losing it to petty fraud, just like supermarkets do with petty theft.

Attack by the seller

You might ask: Can the seller deceive the payer by providing an **addrchap** address space that has more than one "winning" address, i.e. an address to which the seller has a private key? In other words, can the seller have two or more winning tickets in the address space given to the payer? In this case, the chances of a payment are doubled, which also doubles the expectation of the average amount paid.

The answer is: Yes, such an event is possible in theory, but its probability is negligible and can be ignored. Let's take a closer look.

As we know, the payment address in Emercoin is a 160-bit number derived by a cryptographically stable hashing of the public key, and it is uniformly distributed throughout the entire range of possible values. Let's take the average **risk** in the system at 2^20 bits, or approximately 1,000,000 (actual values will be much lower). Then the attacker has 2^140 (140=160-20) **addrchap** options to search and try to generate a pair of addresses for the same **addrchap**. Using a birthday paradox-based strategy, an attacker can solve the pair search task in about 2^70 picks, while using 2^70 memory cells to store keys. Assuming that a modern computer can generate a key in 10 milliseconds, we get the result that finding the pair will take more than 374 billion years. To put that into perspective, the universe is about 12 billion years old. Even if we assume that the attacker has a cluster of millions of computers, each of which is 1000 times more powerful than a modern PC, the search time becomes a "mere" 374 years.

All in all, such an attack is theoretically possible, but organizing it requires tons of time and resources. It's more practical to simply mine Emercoin.

Document author and Randpay creator: **Oleg Khovayko**