```
BIP: 15
Layer: Applications
Title: Aliases
Author: Amir Taaki <genjix@riseup.net>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0015
Status: Deferred
Type: Standards Track
Created: 2011-12-10
```

BIP 0070 (payment protocol) may be seen as the alternative to Aliases.

Using vanilla bitcoin, to send funds to a destination, an address in the form 1Hd44nkJfNAcPJeZyrGC5sKJS1TzgmCTjjZ is needed. The problem with using addresses is they are not easy to remember. An analogy can be thought if one were required to enter the IP address of their favourite websites if domain names did not exist.

This document aims to layout through careful argument, a bitcoin alias system. This is a big modification to the protocol that is not easily changed in the future and has big ramifications. There is impetus in getting it correct the first time. Aliases have to be robust and secure.

Schemes

Here are a few different proposals and the properties of each system.

FirstBits

FirstBits is a proposal for using the blockchain as an address book.

When bitcoins are sent to an address, that address becomes recorded in the blockchain. It is therefore known that this address exists or did exist by simply seeing that there was a payment to that address. FirstBits is a method to have a memorable alias. One first converts the address to lower-case, then takes the first few unique characters. This is your FirstBits alias.

As an example, brmlab hackerspace in Prague has an address for purchasing food or drink, or making donations:

```
1BRMLAB7nryYgFGrG8x9SYaokb8r2ZwAsX
```

Their FirstBits alias becomes:

```
1brmlab
```

It is enough information to be given the FirstBits alias *1brmlab*. When someone wishes to make a purchase, without FirstBits, they either have to type out their address laboriously by hand, scan their QR code (which requires a mobile handset that this author does not own) or find their address on the internet to copy and

paste into the client to send bitcoins. FirstBits alleviates this impracticality by providing an easy method to make payments.

Together with Vanitygen (vanity generator), it becomes possible to create memorable unique named addresses. Addresses that are meaningful, rather than an odd assemblage of letters and numbers but add context to the destination.

However FirstBits has its own problems. One is that the possible aliases one is able to generate is limited by the available computing power available. It may not be feasible to generate a complete or precise alias that is wanted- only approximates may be possible. It is also computationally resource intensive which means a large expenditure of power for generating unique aliases in the future, and may not scale up to the level of individuals at home or participants with hand-held devices in an environment of ubiquitous computing.

FirstBits scales extremely poorly as the network grows. Each indexer or lookup node needs to keep track of every bitcoin address ever in existence and provide a fast lookup from the aliases to those addresses. As the network grows linearly, the number of addresses should grow exponentially (assuming a networked effect of (n-1)*(n-2)/2) rapidly making this scheme unfeasible.

Light clients of the partial merkle root types become dependent on a trusted third party for their alias lookups. The cost of storing every bitcoin address is too high considering their typical use-case on low-resource devices. This factor more than the others, means this scheme is sub-optimal and must be rejected.

DNS TXT Records

DNS allows TXT records to be created containing arbitrary data. In a bitcoin alias system, a custom format mutually agreed upon by a BIP standard would be used to store mappings to bitcoin addresses from domain names. How such a format would look is out of the scope of this document.

An issue is that it requires people who wish to create such mappings to be familiar with configuring DNS records, and be able to run the necessary toolsets to insert the correct data. Although not a huge concern, it is a usability issue.

Security wise, DNS is unsafe and insecure by design. It is possible to spoof records by being on the same network as another host. A number of revisions to mitigate the issue under the guise of DNSSEC have been in the works since the 1990s and are still being rolled out.

As of Dec 2011, DNSSEC is still not yet a defacto standard on the internet. Should a participant in the bitcoin network wish to use DNS TXT records, they would in addition to having to configure DNS, be able to setup DNSSEC. This may not be feasible, especially where some registrars provide access to DNS through a web interface only.

The disadvantage of DNS TXT records is that updating a record takes time. This encourages people to not use new addresses per transaction which has

certain security issues.

Server Service

Aside from using DNS TXT records, another possibility is using the domain name system to lookup hosts and then contact a service running on a predefined port to get the bitcoin address.

1. User wishes to send to foo@bar.net
2. Client uses DNS to find the IP address of bar.net: 123.123.123.123
3. Client connects to port 123.123.123.123:4567 and requests the bitcoin address for the user *foo*
4. Server responds with the address or error code and terminates the connection.
5. Client sends the funds to the address

The service would be responsible for providing the mechanisms for changing and storing the mappings on their service. A front-end web interface could be provided to users wishing to use the service and customise their accounts on the server.

This approach has the positive aspect of providing the best flexibility for the implementer to store the records however they wish in a database or plaintext file, and then serve them up quickly using a small server side daemon typically written in C. This approach is highly scalable.

However this approach also suffers the problem of being reliant on DNS and hence also being vulnerable to spoofing. Hence DNSSEC is also required. This approach is slightly better than the DNS TXT records though since it makes inserting new users and modifying aliases very easy which allows people to run these server services more cheaply.

HTTPS Web Service

HTTPS provides an additional layer of security by encrypting the connection, providing much needed privacy for users. Together with using Certificate Authorities, it fixes the issue with using DNSSEC since an error would be thrown up were someone to try to spoof a domain name on the local network.

When trying to send to:

```
genjix@foo.org
```

The request is broken into the handle (genjix) and domain (foo.org) at the last occurrence of the @. The client then constructs a request that will query for the address.

```
https://foo.org/bitcoin-alias/?handle=genjix
```

bitcoin-alias has been chosen as the query suffix because it allows this system to co-exist easily within another web root without the fear of name clashes.

The query will then return an address which is used to make the payment.

```
1Hd44nkJfNAcPJeZyrGC5sKJS1TzgmCTjjZ
```

The details of whether a unique address is returned per query, whether an address is fetched from a pre-existing pool of addresses, and so on is an implementation detail unique to every server. How alias to address mappings are setup is dependent on the site which could have a web-interface and be providing a free service to users or be a private customised service serving pre-existing addresses. This is left up to sysop policy, and deliberately not defined here.

A web service is trivial to setup and the cost is low. There are many free out of the box providers on the net that allows anyone with the most basic knowledge of web technologies to create their own website. By providing users with a package, anybody can quickly set themselves up with a bitcoin alias. It could be something as simple as a PHP script that the user edits with their custom settings and uploads themselves to their website.

It also scales reasonably- anybody wishing to run a naming service can attach a backend with a variety of database technologies then provide a web frontend for users to customise and create their own aliases.

A naive implementation is provided below as an example.

```
// resolv.h
#ifndef NOMRESOLV_H__
#define NOMRESOLV_H__

#include <string>
#include "curl/curl.h"

using std::string;

/*

This class resolves against a server to lookup addresses.
To not conflict with the bitcoin addresses, we refer here to people's handles.
A handle is of the form:

   genjix@foo.org

Most characters are valid for the username + password (and handled accordingly), but the dom

   genjix@bar.com/path/to/

*/

class NameResolutionService
{
```

```
public:
    NameResolutionService();
    ~NameResolutionService();

    // Three main methods map to RPC actions.
    string FetchAddress(const string& strHandle, string& strAddy);

private:
    // A POST block
    class PostVariables
    {
    public:
        PostVariables();
        ~PostVariables();
        // Add a new key, value pair
        bool Add(const string& strKey, const string& strVal);
        curl_httppost* operator()() const;
    private:
        // CURL stores POST blocks as linked lists.
        curl_httppost *pBegin, *pEnd;
    };

    // Explodes user@domain => user, domain
    static void ExplodeHandle(const string& strHandle, string& strNickname, string& strDomai
    // Perform the HTTP request. Returns true on success.
    bool Perform();

    // CURL error message
    char pErrorBuffer[CURL_ERROR_SIZE];
    // CURL response
    string strBuffer;
    // CURL handle
    CURL *curl;
};

#endif
```

```
// resolv.cpp
#include "resolv.h"

#include <boost/lexical_cast.hpp>

#include "access.h"

// callback used to write response from the server
static int writer(char *pData, size_t nSize, size_t nNmemb, std::string *pBuffer)
```

```
{
  int nResult = 0;
  if (pBuffer != NULL)
  {
    pBuffer->append(pData, nSize * nNmemb);
    // How much did we write?
    nResult = nSize * nNmemb;
  }
  return nResult;
}

NameResolutionService::NameResolutionService()
{
    // Initialise CURL with our various options.
    curl = curl_easy_init();
    // This goes first in case of any problems below. We get an error message.
    curl_easy_setopt(curl, CURLOPT_ERRORBUFFER, pErrorBuffer);
    // fail when server sends >= 404
    curl_easy_setopt(curl, CURLOPT_FAILONERROR, 1);
    curl_easy_setopt(curl, CURLOPT_HEADER, 0);
    curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1);
    curl_easy_setopt(curl, CURLOPT_POSTREDIR, CURL_REDIR_POST_302);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, writer);
    curl_easy_setopt(curl, CURLOPT_USE_SSL, CURLUSESSL_TRY);
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1);
    // server response goes in strBuffer
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &strBuffer);
    pErrorBuffer[0] = '\0';
}
NameResolutionService::~NameResolutionService()
{
    curl_easy_cleanup(curl);
}

void NameResolutionService::ExplodeHandle(const string& strHandle, string& strNickname, stri
{
    // split address at @ furthrest to the right
    size_t nPosAtsym = strHandle.rfind('@');
    strNickname = strHandle.substr(0, nPosAtsym);
    strDomain = strHandle.substr(nPosAtsym + 1, strHandle.size());
}
bool NameResolutionService::Perform()
{
    // Called after everything has been setup. This actually does the request.
    CURLcode result = curl_easy_perform(curl);
    return (result == CURLE_OK);
```

```
}

string NameResolutionService::FetchAddress(const string& strHandle, string& strAddy)
{
    // GET is defined for 'getting' data, so we use GET for the low risk fetching of people'
    if (!curl)
        // For some reason CURL didn't start...
        return pErrorBuffer;
    // Expand the handle
    string strNickname, strDomain;
    ExplodeHandle(strHandle, strNickname, strDomain);
    // url encode the nickname for get request
    const char* pszEncodedNick = curl_easy_escape(curl, strNickname.c_str(), strNickname.siz
    if (!pszEncodedNick)
        return "Unable to encode nickname.";
    // construct url for GET request
    string strRequestUrl = strDomain + "/bitcoin-alias/?handle=" + pszEncodedNick;
    // Pass URL to CURL
    curl_easy_setopt(curl, CURLOPT_URL, strRequestUrl.c_str());
    if (!Perform())
        return pErrorBuffer;
    // Server should respond with a JSON that has the address in.
    strAddy = strBuffer;
    return "";  // no error
}

NameResolutionService::PostVariables::PostVariables()
{
    // pBegin/pEnd *must* be null before calling curl_formadd
    pBegin = NULL;
    pEnd = NULL;
}
NameResolutionService::PostVariables::~PostVariables()
{
    curl_formfree(pBegin);
}
bool NameResolutionService::PostVariables::Add(const string& strKey, const string& strVal)
{
    // Copy strings to this block. Return true on success.
    return curl_formadd(&pBegin, &pEnd, CURLFORM_COPYNAME, strKey.c_str(), CURLFORM_COPYCONT
}

curl_httppost* NameResolutionService::PostVariables::operator()() const
{
    return pBegin;
}
```

```
</source>

<source lang="cpp">
// rpc.cpp
...

const Object CheckMaybeThrow(const string& strJsonIn)
{
    // Parse input JSON
    Value valRequest;
    if (!read_string(strJsonIn, valRequest) || valRequest.type() != obj_type)
        throw JSONRPCError(-32700, "Parse error");
    const Object& request = valRequest.get_obj();
    // Now check for a key called "error"
    const Value& error  = find_value(request, "error");
    // It's an error JSON! so propagate the error.
    if (error.type() != null_type)
        throw JSONRPCError(-4, error.get_str());
    // Return JSON object
    return request;
}

const string CollectAddress(const string& strIn)
{
    // If the handle does not have an @ in it, then it's a normal base58 bitcoin address
    if (strIn.find('@') == (size_t)-1)
        return strIn;

    // Open the lookup service
    NameResolutionService ns;
    // We established that the input string is not a BTC address, so we use it as a handle r
    string strHandle = strIn, strAddy;
    string strError = ns.FetchAddress(strHandle, strAddy);
    if (!strError.empty())
        throw JSONRPCError(-4, strError);

    const Object& request(CheckMaybeThrow(strAddy));
    // Get the BTC address from the JSON
    const Value& address = find_value(request, "address");
    if (address.type() != str_type)
        throw JSONRPCError(-32600, "Server responded with malformed reply.");
    return address.get_str();
}

// Named this way to prevent possible conflicts.
Value rpc_send(const Array& params, bool fHelp)
```

```
{
    if (fHelp || params.size() != 2)
        throw runtime_error(
            "send <name@domain or address> <amount>\n"
            "<amount> is a real and is rounded to the nearest 0.01");

    // Intelligent function which looks up address given handle, or returns address
    string strAddy = CollectAddress(params[0].get_str());
    int64 nAmount = AmountFromValue(params[1]);
    // Do the send
    CWalletTx wtx;
    string strError = SendMoneyToBitcoinAddress(strAddy, nAmount, wtx);
    if (!strError.empty())
        throw JSONRPCError(-4, strError);
    return wtx.GetHash().GetHex();
}

...
```

IP Transactions

An IP transaction is an old transaction format in bitcoin that is disabled and possibly could be deprecated. It involves being given an IP address to make payment to. Upon connecting to the node and requesting their public key using "checkorder", they will respond with a script in the form:

```
OP_CHECKSIG
```

Similar to coinbase output transactions. IP transactions have the advantage of being able to contain additional metadata which can be useful in many transactions. Currently no authentication is done making the scheme insecure against man in the middle (MITM) attacks.

This proposal seeks to enable DNS lookups for IP transactions.

The "checkorder" message would contain a destination account, which could map to different isolated sets of keypairs/wallets running under the same host. The exact mapping from the checkorder reference info to the local system is implementation defined.

By using DNS lookups, the MITM problem with IP transactions could be mitigated by storing a public key in a DNS TXT record. This public key would be used for all future "reply" messages originating from that host. First time use would require a confirmation for acceptance of that public key; like with SSH. Should the "reply" message not match the accepted public key, then the host will be given an error.

Namecoin ID

This proposal uses the Namecoin blockchain to associate an alias with a bitcoin address. Bitcoin queries a namecoin node. This retreives the structured data containing the bitcoin address(es) associated with this alias.

Using a decentralised domain name system like Namecoin, means no external server or entity needs to be trusted unlike the other proposals listed here. This indicates a system with the advantage of having a high availability and ease of entry (no restrictions for users to create aliases).

Two examples are presented below. The first shows a simpler format, while the second shows several Bitcoin addresses in a structured format.

```
$ namecoind name_show id/khal
{
"bitcoin" : "1KHAL8bUjnkMRMg9yd2dNrYnJgZGH8Nj6T"
}
```

```
$ namecoind name_show id/khal
{
"bitcoin" :
{
"default" : "1KHAL8bUjnkMRMg9yd2dNrYnJgZGH8Nj6T",
"donation": "1J3EKMfboca3SESWGrQKESsG1MA9yK6vN4"
}
}
```

More possibilities :

- Allow to securely use **unsecured channels**

You can put an url and a bitcoin address that will be used to sign the result. It means that a query to this url will return a bitcoin address and a signature. Bitcoin can then check (with the verify_message function) that the returned address has not been replaced by another one.

```
$ namecoind name_show id/khal
{
"bitcoin" :
{
"url" : "http://merchant.com/bitcoin/getnewaddres/",
"signedWith" : "1KHAL8bUjnkMRMg9yd2dNrYnJgZGH8Nj6T"
}
}
```

- Allow to get a different address each time, or per user, per order, etc

```
$ namecoind name_show id/khal
{
"bitcoin" :
```

```
{
"url" : "http://merchant.com/bitcoin/getaddres/{Your customer id}",
"signedWith" : "1KHAL8bUjnkMRMg9yd2dNrYnJgZGH8Nj6T",
"useOnce": false
}
}
```

In the above example, bitcoin will ask the user for "Your customer id" and replace that value in the url before making the http request. The merchant will receive the request and give the user a payment address associated with that customer.

Any text can be put into the brackets, allowing merchants to adapt it to all their needs.

- Specification is extensible

New features can be added later to support uncovered cases.

See the specification of Namecoin ID for more informations.

E