

# Decall: A decentralized, secure, and open-source phone & messaging system

White paper

ALPHA

2021 12 17

<https://decall.org>

Written by Ben Stokman

[ben.stokman@protonmail.com](mailto:ben.stokman@protonmail.com)

764E 817B DB60 3738 EFF9 F90A 7EC9 6BBB 6B95 B638

**Please note:** I refer to the three current methods of calling, SMS, and MMS as “the phone system” since there really isn’t a good name for it.

# Why the current phone system needs to be replaced

I hate the current phone system the world uses today. It’s insecure, expensive, and slow.

It’s always bothered me that there isn’t Internet-based system for calling and messaging; so much so that I’ve been designing Decall on and off for about five years now. I’ve recently become increasingly annoyed at the phone system’s persistence, so I’m moving Decall’s creation up in my priorities, starting with the creation of this white paper, and thus the design and specification of the protocol.

## Expense

A home phone plan costs around \$25 a month in the U.S. Not a mobile phone one, a home phone plan, even ones that use the Internet as a backbone.

Compare that to email, which is typically free. I’ll let that speak for itself.

## Security

There is none.

Okay fine, there’s *some*, but there might as well be none.

The fact that number spoofing is even *remotely* possible (there are measures in place to prevent it now, but the FCC had to hound carriers to support it) is absolutely appalling.

Email has had DKIM for a decade, which is flawless at detecting when a message isn’t sent from an authorized server. And almost all email servers support SSL for messages in transit to avoid eavesdropping.

Meanwhile, I couldn’t even find a straight answer on how calls, SMS, or MMS communications are secured. I assume that cellular networks have some sort of encryption, even if it should be assumed to be crap, but I don’t think there’s any encryption beyond that.

What’s even more worrying is that almost all communications go through the Internet. I’d bet that in order to save costs, carriers don’t bother with updating extremely old protocols, since they

would have to coordinate with all the other carriers, replace equipment and update servers, and train technicians. As I said, the FCC had to hound carriers to support whatever stops call spoofing nowadays.

## Speed

I constantly get messages like, 5 to 10 minutes late. I don't know why specifically, but it's definitely . Just yesterday my friend told me in *the chat of a Minecraft server* to listen to a voice message he sent me, only to find that it got delivered 3 minutes after he sent it.

Even calls have significant delay; I'm talking like, a solid second for the same area code.

## Decall At A Glance

The Decall protocol is actually surprisingly simple; it's just a bit of finesse built on top of HTTP, PGP, and JSON.

There are two separate processes involved with Decall. Because a phone system necessarily requires that there always be a receiving server on and ready to receive data, I have split the protocol into two, with one half specifying how a message is composed and sent to one of these servers (due to the end-to-end nature of Decall, a message can be sent from anywhere, unlike email which necessitates it be sent from specific servers for verification purposes), and another half being how a client (phone, application, etc.) would communicate with these servers in order to download, sync, and optionally send messages.

## The Reasoning Behind the Parts

As I said before, Decall is built on top of HTTP, PGP, and JSON. In this section, I will explain my reasoning behind using these.

### HTTP

All Decall communication will be through HTTP as opposed to a custom protocol. This provides many benefits:

1. Providing an easy extra layer of security through SSL to help prevent eavesdroppers from seeing who is communicating with whom.

2. Making it almost ridiculously easy and straightforward to create applications that work with the Decall protocol.
3. Making it way harder to censor & easier to bypass censorship.
4. It is significantly more stable.
5. I don't have to do as much work.

## PGP

A personal criticism I have with other end-to-end chat applications is that they use a specialized encryption method not shared with anything else. I'm not necessarily saying it's insecure like this, it's just that there's very little way for the end user to open up this encryption to inspect it. And even worse, the security is usually entirely reliant on some central server, with no way for the user to manually input an encryption key.

PGP is perfect for Decall. It's extremely robust, and already ubiquitous with email. I even use the timestamping function as the way to know when a message was sent, completely removing any responsibility from the server to manage timestamps.

Decall's design doesn't require that an end-user manage encryption keys in any way, but it does allow for them to. It's an excellent trade-off between ease-of-use and security. The complexity of the end-to-end encryption is hidden unless the user wants to see it.

## JSON

In order to hide all that complexity, Decall uses JSON objects to define the necessary things about a number. I chose JSON due to how easy it is for both humans and computers to read.

The following is an example JSON number data object:

```
{
  "number": "john.doe@example.com",
  "name": "John Doe",
  "contactcard": "<vcard data>",
  "type": "personal",
  "timeout": 30,
  "key": "<ASCII-armoured PGP Key>",
  "keyfingerprint": "<key fingerprint>",
  "oldkeys": [
    {
      "fingerprint": "<key fingerprint>",
      "burnt": false
    }
  ]
}
```

```

    }
  ],
  "methods": [
    {
      "label": "Main",
      "priority": 0,
      "address": "https://decall-server.example.com",
    },
    {
      "label": "Tor",
      "priority": 1,
      "tor": true,
      "address": "http://<onion-address>.onion",
    }
  ]
}

```

## Explanations for Each Property

- “number” is a simple label for what the actual number is supposed to be. This can be used for clients to keep track of cached number data, as well as easier in-person exchange of number data (for both better security, and for adoption purposes).
- “name” is just the name of the person. This is outside the optional VCard data.
- “contactcard” is optional Vcard data; use “\n” for line endings.
- “type” is used to indicate if the number is personal, home, or a company. This is pure semantics and is only useful to the end-user since any computer can just extract the fingerprint from the key itself.
- “timeout” is how long in seconds a caller should wait for the receiver to pick up. Default is 30.
- “key” is an ASCII-armored PGP key that will be used to encrypt all communication; use “\n” for line endings. A key is absolutely required.
- “keyfingerprint” is used for humans to easily see what the current key’s fingerprint is.
- “oldkey” is an array of old key fingerprints, this is used to verify the authenticity of old messages, since the current key may have changed due to the old one being expired or burnt.
- “methods” is an array of possible methods that can be used to communicate with the number.

- “label” can be anything. This is almost never used by the end user, as the method utilized is usually an automatic process
- “priority” is used by clients for an order to try and make a successful connection. If the priority 0 method fails to connect and there is a priority 1, the client will try using it if possible, and so on.
- “tor” is used to indicate if the method requires the Tor network to access the method’s server.
- “address” is where the server can be located at. The underlying protocol will always be HTTP(S), even over Tor.

## Creating, Sending, and Verifying a Message

All PGP stuff is ASCII-encoded.

1. The sender downloads the JSON object for the intended recipient(s). These objects can be found at `http(s)://<domain>/<email>.json` for email numbers, and a TXT record with “decall-number” as the name for domain numbers.
2. The sender then signs and encrypts the data just as normal for PGP, and includes themselves as a receiver too.
3. The sender then signs and encrypts some metadata and appends it to the end of the encrypted data. The sender includes themselves as a receiver for the encrypted data. The signature vitally must have the same timestamp as the one for the data itself. The metadata included is as follows, with each field separated by a line ending strictly in this order:
  1. The 4 digit message ID – used for multiple messages in the same second as well as piecing together calls.
  2. 32-bit HEX Conversation/message ID.
  3. A comma-separated list of intended receivers (does not include sender).
  4. An adler32 checksum of the data.
  5. The message type (call, call-handshake, file, text).
  6. The Media/MIME type of the data.
  7. (Optional) the filename of the data.

4. The sender then signs the concatenation of steps 2 and 3, and appends it to the end. This data is not encrypted.
5. The sender then takes the completed data and submits an HTTP POST request to a URL with the following pattern: <the server URL>/send?email=<receiver email>&sender=<sender email>. If there are multiple receivers of the message, the sender will send multiple POST requests, and can optionally include multiple emails separated by commas.
6. The receiving server will then fetch the sender's JSON object to verify the signature created in step 4, and thus that the sender is . This is done for anti-spam purposes.

## Interacting With Servers

This part is a bit more complicated.

Basically, I understand that people will want to use Decall servers that they don't have to run on their devices, mainly because they might want to use multiple devices. I also want to avoid a scenario where there's a standard war for how data gets delivered to the end user. The whole point of this is to be an open protocol.

There are 4 different actions:

Auth

Send

Receive

Delete

### Auth

This is used to get and manage authentication cookies. These cookies are 256 bits of hex data, randomly generated on the server.

### Creating a Cookie

1. The client sends a GET request to <server URL>/auth?action=requestcookie&user=<email address>. The server responds with a JSON, with "user" being true or false, corresponding to if the user exists, and, if "user" is true, "cookie" being a string in the format of "decallauth<256 hex bits>".

2. The client then signs this data with the proper PGP key, then sends a POST request with the data to `<server URL>/auth?action=authcookie?user=<email address>`. The server will respond with a JSON with "success" either true or false. The cookie does not contain "decallauth"; that is just there so the signature doesn't get confused for something else.
3. The client can also test this cookie at any time with a GET request to `<server URL>/auth?action=testcookie?user=<email address>` whilst submitting the cookie with the request. The server will respond with a JSON with "success" being either true or false.

## Deleting Cookies

I highly recommend that servers automatically delete cookies that haven't been used for about two weeks or so, after all, a user can just request a new one. But in case something bad happened, I have included the option to delete all cookies by GETting the following URL: `<server URL>/auth?action=purgecookies` with any valid cookie. The server will respond with a JSON with "success" either true or false, corresponding to if the action was completed or not.

Send

Receive

Delete

## How Calls Work