flow Documentation (experimental)

Manfred Morgner

September 9, 2011

Contents

1 *flow* **1**

1.1 What *flow* is . . . 1

1.1.1 What *flow* does . . . 1

1.1.2 Why *flow* is developed . . . 2

1.1.3 What *flow* is good for . . . 2

1.2 Some security basics . . . 2

1.2.1 Two simple definitions . . . 2

1.2.2 Trustworthyness . . . 3

1.2.3 To trust or not to trust . . . 3

1.2.4 Identity in open networks . . . 4

1.2.5 How to trust . . . 5

1.2.6 Watch your ways! . . . 6

1.2.7 Why should you care? . . . 6

1.2.8 Where to go from here . . . 7

2 The project **9**

2.1 Concept . . . 9

2.1.1 The 'man in the middle' . . . 9

2.1.2 The message server . . . 9

2.1.3 Data protection . . . 10

2.2 Secondary targets . . . 10

2.2.1 Training . . . 10

2.2.2 Education . . . 10

2.2.3 Knowledge 10
2.3 Source Code 10
2.3.1 class Pulex 10
2.3.2 class Crypto 12

3 Quickstart API description **15**
3.1 No library, no programming dependencies 15
3.2 Transport Layer 15
3.3 Addressing 16
3.4 Data Transport 16
3.4.1 Data transport packaging 16
3.4.2 Data to send 17
3.4.3 Data to receive 17

Data must *flow*!

Chapter 1

flow

flow will move your data in protected mode

1.1 What *flow* is

flow is a client server application system which may move your data securely. It is not necessary, but you might use insecure networks and also untrusted service providers with *flow* without breaching your privacy. The level of security you get out of *flow* is up to you! Meaning you have nearly unlimited control over your data whilst they are on their ways. At least, if you're the recipient too - which is not as crazy as it sounds. *flow* will also support you in establishing privacy whilst communicating with trustworthy partners.

It is important to understand, that there is nothing in the whole world that will protect your privacy as long as you're communicating with others. The only way to guarantee your privacy is not to communicate, even with yourself!

YOUR data is data you own, data you send and data you receive. Receiving data may imply third entities. Other people for example, or data gathering devices.

As far as you're in control of all the security credential at both ends of a transportation chain, you are in control of your data. Other people, friends for example, will undermine your control over your data but only in limited ways. I will dig deeper into this later on.

1.1.1 What *flow* does

flow secures your communication with dedicated partners as far as possible and does not make nonsense down the road. This way it maximizes the privacy you get while communicating with remote partners and reenforces you to become your own master of your privacy.

Further, *flow* will become able to use 'foreign transportation' such as email, chat and other means of 'well known' mechanics. For this, *flow* needs to be as slick as possible, meaning, it

is to be reduced to absolute minimum in respect of functionality and complexity. So it becomes easy to encapsulate it in other transport environments.

1.1.2 Why *flow* is developed

flow is developed for some different reasons

- A young man and a women I know didn't believe I'm able to program in C++ and (possibly) other languages. I need to put this to test.
- There are no useful communication applications ensuring my privacy.
- I was in need of some tools dealing with OpenSSL.
- I couldn't believe that OpenSSL is as complicated as it is. Now I'm a believer.

1.1.3 What *flow* is good for

flow solves the problem you enter if you need to communicate and wish the communication to be protected. You may wish to establish separate channels of communication for different communication entities - which may mean people, friends, devices, ... you name it, *flow* claims it.

Further you may adopt different identities for different communication partners or don't wish to have an identity at all - at least in certain cases. *flow* will enable you to do so.

Indeed, having no identity is the normal operation mode for *flow* . Having one is an enhancement.

1.2 Some security basics

1.2.1 Two simple definitions

Assuming that data you send to others (autonomous entities, not controlled by you) are not truly your data, we have to split the analyse in two parts:

YOUR data: Means data over which you have full control and which will not change their ownership.

THEIR data: Means data you share and therefore need to be given some control into others hands.

1.2.2 Trustworthyness

Nearly no one knows about Software Certificates. Good willing people, business man, manager and lot of other people tried to send the message about Software Certificates (for our purpose simply 'Certificates') to the world, but the world was not listening. This is because the matter is much too complicated.

Because of this, I will not try to do the next boring attempt to explain. If you'r not familiar with the matter and - as usual - not interested at all, forget the whole chapter about security. You don't need to care about it. Other ones care for you. *The last one is a joke!*

One key question in all matters of security is: Who is trustworthy?

The answer seems to be simple but it is not. You may take a most restrictive point of view and state, that the only person you are will to trust is you. But given the point of our discussion, an open network and (per definition) not trustworthy service providers building this network, your believe has to be wrong.

You'r only trustworthy against yourself if you understand the basics of data security. Even if you read this document, which means you'r interested in Data Security, probably you should not trust yourself - at least for the beginning.

1.2.3 To trust or not to trust

Should you trust yourself? Why? What is it you have or know or believe, that should make you trust yourself? Watch yourself for a while. Do you use cellphones in public? Did you ever wrote an SMS while using public transport? Did you talk about business matter or partnership issues in public? No? Ok, you may be trustworthy against yourself. Yes? You may think about information leakage for a while.

Should you trust others? I don't believe so and you shouldn't too. This is because of two reasons.

- You don't know if your communication partner knows about security.
- You don't know if your communication partner put his/her (maybe) knowledge into action.
- You don't know if your communication partner changes his/her mind after some time.

As you can see, each time you think about it, there are more reasons not to trust than you believed at the beginning!

A typical scenario is the erotic photo you sent to your friend. If your relationship breaks, your friend may decide to - accidentally - let the picture slip into to open. It might give you a kick to think about the question WHY this can happen.

The normal answer would be: *You were careless*. Another one may be: *This people was not trustworthy.* But both are not the real answer. The really real answer is:

"Because you gave the control over your data out of your hands."

Some may love this, but hopefully you get a better feeling if I paint a picture of what *giving control over your private data away* means:

You're blindfolded. Your friend leads you for at least an hour through a foreign landscape. The last twenty minutes the way appears to go through a dense wood. After this, he/she strips off your cloths, ties you up, wide spread between some trees removing your blindfold. In a distance you here car doors closing.

Some people may find this situation comfortable. Most of us would stop the performance at a certain point. But why?

"Because we don't trust our friends enough with our privacy."

And if we do, possibly we shouldn't. At least with undeniable insides of your privacy.

Compared with entrusting private data to others, this pictures is indeed harmless! If we try to transfer this situation into the world of information, we do not only speak about pictures, private ideas and privacy, we also speak about identity. One of the simpler form of identity theft is thieving credit card information. But even this may lead your way directly to jail. For example after some one used your private credit card information to buy illegal products. Such events may lead to some simple collateral damage like some weeks in jail and some month amongst layers. But this all is really a joke compared with what is possible if someone really steals your identity.

Even if these thoughts lead a bit too far out you should keep them in mind. There are objective information, als pictures, and subjective ones, as tests. Luckily for all of us, as technology progresses through time, more and more of such information become deniable.

But in some cases, you simply wish some information not to leak out before a certain time period. This is the moment when *flow* comes handy.

1.2.4 Identity in open networks

Who are you?

Sorry, I'm not interested, but believe me, someone is. A lot of people wish to know your better, wish to know exactly who you are. Sales man are the least frightening ones. You may believe that thiefs and other criminals are not interested in your identity because you don't know any, but you're wrong. Internet criminals are interested in everyone whom they are able to uncover. You leave traces and lots of individuals, organisations and governments are investing in tracing people, uncovering them and get as much social and other kind of information as possible.

The use of such information is countless. It would fill some books for its own.

But sometimes you run into greater problems. You need to prove your identity against a communication partner but hide it from everyone else. And some times not only this, some times you wish only your communication partner knows that you are the right person but not really who you are.

This may be an extreme case, but implementing secure, reliable private communication needs to put the possible requirements to the limit. If you've lesser requirements, dealing with them will become easier than dealing with the worst case and may result in some protection you would not get if the limit is only what you currently imagine.

For the wise: IP address issues are in no case a discussion point in this document. *flow* is completely ignorant and transparent against IP address fiddling.

1.2.5 How to trust

If we assume trusting each other is recognizing each other for what one promised to be for the other one, it is in no way easy. One of the easier solutions is using an well know communication channel to exchange the necessary enhanced authentication credentials.

For example, you are emailing with a partner for a while, so you may send him your Certificate the same way; by email. This looks simple and indeed it is. But the receiver cant be certain that this Certificate belongs to you indeed. Maybe, someone else interfered. Thats why you may exchange other prove using other ways of communication.

For example, you may print your Certificate and send it by snail mail or hand it over directly. There are different methods. But do not trust any 'web of trust'! In a so called 'web of trust' everyone believes every other one to be careful. A big mistake if you try to establish private communication.

For example: Calling a certificate from a public repository may give a listener (log file reader, ...) a hint you might be in need of private communication with the person/entity/unit the certificate is signed to.

You may trust a web of trust to be reliable if you need to communicate with partners with your and their identity openly known. This should be the case if you do public business with them. But if you wish to be anonymous for the public but also sure to know with whom you're communicating with, you fail to fulfill two basic rules for a 'web of trust'.

At first, your Certificate does not confirm you by its public content. And second, letting others know who you're for later prove, you discover your identity against others and so (eventually) discover it to others who wish to know who you are but shouldn't reach this conclusion.

The best way to ensure others don't know who you are is to ensure, others don't know who you are!

1.2.6 Watch your ways!

Last, but surely not least, you have to watch your way, or speaking clearly, the traces you leave behind. If you deal with a set of real people in the real world (we call this 'Set R') and a set of virtual people in the virtual world (we call this 'Set V') then you might be in for an enhanced 'Set Mathematics'.

Possibly you don't think about it, but if you leave your certificates in relation to other certificates on some untrusted servers, watchers may - at the beginning - don't know who is communicating with whom, these watchers are able to spin a web of communication connections simply by watching which certificate becomes associated to which others.

Given the assumption that communication behaviour in Set R becomes reflected in Set V, an onlooker may conclude who is who in your web of virtual entities. Using *flow* , you may break the mathematical set by a simple measure. Use different certificates to communicate with the same partner.

You should have a fluid set of certificates to communicate with your Set V and you should not use the same certificate to communicate with different partners. You even may not use the same communication server to communicate with different partners - if possible. Break the set in as many parts as possible!

To confuse the observer, you even should relay random posts from others for others. Which distorts the truth about observed communication patterns if IP addresses are part of the observation. Also it might be helpfull if you send out messages of no true content to members of Set X, from whom you possibly don't know anyone.

For an uninvited observer, an untrusted service provider this would create a buzzing where observers will be filled up with useless traffic, false connections and misleading traces.

Even if an observer becomes part of the conversation, as long as you don't send personal data with your useless buzzing messages, he will not gain useful information out of such measures.

1.2.7 Why should you care?

You don't know what will happen.

Friends become foes. So, possibly you wish to deny that you're the communication partner after a former friend went against you.

Communication partners may be accused to be involved in some criminal act you don't wish to become associated with.

You don't want to spread you social net - Set R - throughout the internet.

You may have some of millions of other reason to care and if you have none, do it anyway. As time comes you will find out why this was a good decision.

1.2.8 Where to go from here

May be someone is kind enough to add a file splitter to the system so you become able to spread a file through the net in parts.

May be there will come a time when your request to your messages is answered be a server only if you come from a certain set of system names, device classes or IP addresses.

And may be it's useful to pull the plug on your messages. Giving the command to delete your messages from the servers you used. This will not help against fraudulent servers but possibly against successful R-world or V-world attacks against trustworthy servers.

Chapter 2

The project

2.1 Concept

The concept of flow is, to move data securely through insecure networks using untrusted providers. To do this, *flow* has to fullfill three requirements

1. Kick out the 'man in the middle'
2. Ensure even the message server is unable know who is who
3. Ensure only the receiver is able to read the data

2.1.1 The 'man in the middle'

The man in the middle is the easier part. He does not love Transport Layer Security - TLS (former SSL) - and usually will focus on other means to break down your defences if he discovers such crud methods of securing your doings. Currently and in the near future there is enough unencrypted, unprotected traffic to read. So its unlikely to become overheard if using TLS.

2.1.2 The message server

But how to provide the communication server with no information of who you are and to whom to deliver the package without telling him who you are and to whom to deliver the package?

Well, this is not only easy, but eases the rest of the boring transportation requirements too. This is, because you neither have to tell the server who your are nor who the receiver might be. A simple two step trick ensures this:

1. You will be (anonymously) identified by your Certificate used in your TLS connection.

2. You address your message by using the identification code of the Certificate of your partner.

For the server (and anyone interfering at the server) this will look like this:

```
FROM:FB0FB87C6F783B35F5F21B69F79F4DCC4EDF1963
TO..:BA95B648CE28F80842CECDC6A4BA8826B6B9FE32
```

Such addressing makes things obscure for an uninvited observer. At least it clearly is not easy to follow conversations if connections made using non personalized addressing like this. One may follow IP addresses but this is of limited use if users are careful.

2.1.3 Data protection

Ensuring that only the receiver will be able to understand your message is easy too. Because asymmetric data encryption is 'state of the art', even if no normal user understands how it works, it ensures exactly this.

The credentials we already need for the previous steps are enough to encrypt the messages for the receiver. These credentials are:

- Our own Certificate, used to establish an encrypted connection to the server.
- The partner Certificate, used to decrypt received messages and to address sent messages.

2.2 Secondary targets

2.2.1 Training

2.2.2 Education

2.2.3 Knowledge

2.3 Source Code

2.3.1 class Pulex

The Pulex class represents a neutral object containing client data to become moved to other clients using a server able to move Container objects. To be able to do so, a Container is able to move data wiht minimum amount of knowledge of what these data are. This works similar to a

packet moved by a post office. The sender ist known, the receiver ist known but the content is not.

Regarding a Pulex, the sender ist mainly unknown, the receiver will become partially known if and when he calls is Container. The server will give the packet to every receiver who successfuly claims to be the receiver of it. To make things with a Pulex not as easy as with a Container, a Pulex sends the identity of its sender and its receiver in the form of a 40 character ascii representation of the fingerprint of their certificates.

The parent class to be used as 'inherited::methode()' for abstract usage of inherited methods. Mimimizes logical redundancy

```
    typedef CContainer inherited;
```

May be this is not a good idea at all, but enables compact code, which also may not be good idea. This operator appends data to the object as if it were a data sink which it is ;-)

```
            const std::string& operator << ( const std::string& rsData );
```

Friendly operators to pipe data to different types of stream

```
    friend std::ostream& operator << ( std::ostream&, CPulex& );
    friend      CSocket& operator << (      CSocket&, CPulex& );
```

A template to be used by output pipe operators to send the puley off

```
    template<typename T>
      T& Send( T& roStream );
```

```
#ifndef _PULEX_H
#define _PULEX_H

#include
#include

#include <list>
#include <iostream>


extern bool g_bVerbose;

class CPulex : public CContainer
  {
  private:
    typedef CContainer inherited;

    static const std::string s_sClassName;

  public:
```

```
            CPulex();
    virtual ~CPulex();

    virtual const std::string& ClassNameGet() const;

            const std::string& operator << ( const std::string& rsData );

    friend std::ostream& operator << ( std::ostream&, CPulex& );
    friend      CSocket& operator << (      CSocket&, CPulex& );
  protected:
    template<typename T>
      T& Send( T& roStream );

  protected:
    long ClientSideIDGet();

  }; // class CPulex


#endif // _PULEX_H
```

2.3.2 class Crypto

```
/***************************************************************************
 crypto.h  - description
 -----------------------
 begin                : Thu Dec 02 2010
 copyright            : Copyright (C) 2010 by Manfred Morgner
 email                : manfred@morgner.com
 ***************************************************************************/

#ifndef _CRYPTO_H
#define _CRYPTO_H

#include

#include <string>
#include <vector>

#include <openssl/rand.h> // RSA
#include <openssl/evp.h>  // EVP_MAX_KEY_LENGTH, ...


extern bool g_bVerbose;

#define RANDOM_BUFFER_SIZE 1024

typedef std::vector<unsigned char> CUCBuffer;
```

```
class CCrypto
  {
  protected:
    CUCBuffer m_oData;

    X509*     m_pX509;      // m_oEvpPkey frees this pointer
    CEvpPkey  m_oEvpPkey;
    CRsa      m_oRsa;

    const EVP_CIPHER* m_pfCipher;
    const EVP_MD*     m_pfDigest;

    unsigned char m_aucKey [EVP_MAX_KEY_LENGTH]; // 32
    unsigned char m_aucIv  [EVP_MAX_IV_LENGTH];  // 16
    unsigned char m_aucSalt[PKCS5_SALT_LEN];     //  8

    static const std::string s_sDelimiter;

              CCrypto(){}
  public:
              CCrypto( const std::string& rsInput );
    virtual ~CCrypto();

    void operator =    ( RSA* pRsa ) { m_oRsa = pRsa; }
         operator RSA* ()            { return (RSA*)m_oRsa; }

    void RsaKeyLoadPublic ( const std::string& rsFileRsaKey );
    void RsaKeyLoadPrivate( const std::string& rsFileRsaKey );
    void RsaKeyLoadFromCertificate( const std::string& rsFileCertificate );

    std::string      EncryptToBase64  ();
    const CUCBuffer& DecryptFromBase64( const std::string& rsBase64 );

  protected:
    void RandomSeed();
    void RsaKeyGenerate();

    void RandomGet( CUCBuffer& roBuffer );
    void RandomGet( unsigned char* pucBuffer, size_t nBufferSize );

    std::string      ConvertToBase64  ();
    const CUCBuffer& ConvertFromBase64( const std::string& rsBase64 );

    std::string ConvertToBase64( const unsigned char* pucData, size_t nSize );
    std::string EncryptToBase64( const unsigned char* pucData, size_t nSize );

    bool EncryptRsaPublic ( CUCBuffer& roBuffer );
    bool DecryptRsaPrivate( CUCBuffer& roBuffer );

    std::string SymetricKeyRsaPublicEncrypt();
    bool SymetricKeyRsaPrivateDecrypt( const std::string& rsEncrypted );
```

```
    int  SymetricKeyMake( const EVP_CIPHER* pfCipher = EVP_des_ede3_cfb(),
                          const EVP_MD*     pfDigest = EVP_sha1() );

  }; // class CCrypto

#endif // _CRYPTO_H
```

Chapter 3

Quickstart API description

3.1 No library, no programming dependencies

There is no library, the complete API consists of communication and data formatting specification. At first you have to establish an encrypted connection to the server, secondly you have to send or call your message(s).

There is no preferred programming language, no specific encryption library and no specific 'inner formatting' of the data. All you need is

1. An encrypted TCP/IP session
2. A client and a server to support the protocol
3. Clients who match each others requirements

As described before, you may use any *flow* server you and your communication partner(s) may reach. But also you are able to deliver whatever you need to as long as your packages fit the transport and container specification.

3.2 Transport Layer

A *flow* server is expected to for the client to use TLS. Client are expected to only communicate with servers using TLS.

For his own protection server are allowed to restrict access be rules as

1. Allow all clients with client certificate - least requirements

2. Allow dedicated client certificates
3. Allow client certificates from specific CA(s)
4. others as TLS enables them

3.3 Addressing

Now switch on your 'abstract thinking mode'!. What is a *flow* Partner?

A *flow* partner is a technical entity. It is the combination of a client certificate and a client software. The certificate has to have the attributes

1. Client authentication
2. Encrypt/Decrypt messages
3. Sign messages

or none at all. (which is the better variant). Thats all! It is best if the certificate does not contain any personal truthfull or otherwise retraceable information. So you may be "Marc Zuckerberg" for the rest of the world, living in "Kairo", "Kenia".

A certificate represents a *flow* partner. If you use a seperate certificate with each of your communication partners, lets say eight of them, you have an eight fold personality. You represent eight *flow* partners.

3.4 Data Transport

3.4.1 Data transport packaging

Currently the data transport uses a simple sequential text transmission. A server builds message containers at his liking for storing messages in a way that enables him to answer queries from the client to list or deliver messages.

A multi message sequence is build like this:

```
r:<recipient> # first message
<other bytes>
r:<recipient> # second message
<other bytes>
.             # end of sequence sign terminated by \r
```

The line delimiter will always be `0x0A`. In the case of message reception `recipient` will be replaced with `sender`.

3.4.2 Data to send

A complete message from a client to a server, called `[client-message]`, is constructed by the following components.

```
r:<recipient>
x:<message data>
```

The answer to such message, to send back from the server to the client, is the server unique message id. A *flow* server has to generate unique message ids for the transport context. Content and structure of such ID is not specified, but it has to fulfil three requirements:

1. It has to be unique for the client
2. It has to be transported in ASCII format
3. Its transport form must not exceed 512 ASCII characters

Said this, it is perfectly in order to use simple enumeration for each client and send the ID as hexadecimal string.

3.4.3 Data to receive

A complete message from a server to a client, called `[recall-message]`, is constructed by the following components.

```
a:<sender>
m:<message-id>
n:<next-message-id>
x:<multiline data>
```

`[next-message-id]` is valid for the context of the recall request. If the client uses a filter, this filter is applied to the answer too. For example, if the client only calls the messages from a certain `sender`, then `[next-message-id]` is the ID of the next message from this sender.

1. If no next message exists, the parameter is left out.
2. If more than one message is delivered but not all for the current context, `[next-message-id]` is only delivered in and for the last package.

The recipient line may appear multiple times for multiple recipients. But if so, all of them has to follow one to each other without any other element between. Because the recipient element starts a new message if following any other element as another recipient element. It may not be advisable to create multi recipient messages but should be possible anyway.

Command	client => server	server => client
add message	r:<recipient> x:<body>	m:<message-id>
change message	m:<message-id> [r:<recipient>] [x:<body>]	
remove message	r:<recipient> m:<message id>	
list mine	c:mine	m:<message-id> [m:<message-id>] [m:<message-id>]
list senders	c:senders	s:<sender-id> [s:<sender-id>] [s:<sender-id>]
list messages	c:messages [s:sender-id]	m:<message-id> [m:<message-id>] [m:<message-id>]
call message	c:get [s:<sender-id>] m:<message-id> [m:<message-id>]	s:<sender-id> m:<message-id> x:<body> | none [s:<sender-id>] [m:<messageid>] [x:<body> | none] n:<message id> | none
call next message	c:next r:<message id>	s:<sender-id> m:<message id> x:<body> n:<message-id> | none

Table 3.1: *flow* Client Commands

The server registers the sender ID (the fingerprint of the used TLS certificate) to the soup. This makes for an unique server ID which may be constructed like this:

```
server-id:sender:recipient:receive-time:context-sequence-id
```

this may look like this:

```
SENDER   :RECIPIENT:RECEIVE-TIME  :SID
0A12D4C9E:B8F02E807:20110325103527:3
```

Fingerprints of keys are longer, the representation above is only for descriptive use!

As a suggestion the client may store the message-ids preceded by their servers identification so it becomes easy to remember on which server which message resides:

```
SERVER-ID:SENDER   :RECIPIENT:RECEIVE-TIME  :SID
C48E35A72:0A12D4C9E:B8F02E807:20110325103527:3
```

It is an important principle of the system that clients are aware and able to deal with an unlimited amount of servers.

But the client is free to do what he wants! There is only one definition and this is the communication protocol. Clients and servers are free to handle their business as they want as long as this enables them to correctly follow the communication protocol while fulfilling their mission.

Further clients should store their sent messages locally, remotely or not at all. Storing them is a complete new chapter and has to be handled with care.

——— END OF TEXT ———

Do not believe anything from here on!

——— END OF TEXT ———

Mainly the order of elements is irrelevant. But "s:" starts a new container. To close the data stream, how may contain multiple containers, the client has to send a line containing a single ".".

Messages from a server to a client is constructed like this:

```
s:<sender>
r:<recipient>
m:<message id>
x:<multiline data>
.
```

where the "r:" line represents the local unique identifier of the message on the server side. The client asks the server using a recall block:

```
c:<recipient>
.
```

List block for messages

```
c:messages
.
```

List block for senders

```
c:senders
.
```

Last read block

```
c:next
l:<server local id of the last received message>
.
```

Receive a specific block

```
c:<server local id>
.
```

Receive all messages

```
c:all
.
```

Receive all messages starting with the next after a a specified one

```
c:all
l:<server local id of the last received message>
.
```

OpenSSL

GPL