tNETacle

Technical Documentation

This document is intended to provide technical information about the tNETacle and its components. It contains the public API documentation and the description of the common mechanism powering the tNETacle. Any developer should have read it before contributing code to the project.

(|€~

Title	Technical documentation
Date	17/01/2013
Authors	tNETacle
email	team@tnetacle.org
Subject	This document is the technical documentation of the tNETacle project. It targets anyone who wish to join us on the tNETacle development.
Version	9

Version	Date	Author	Section	Comments
0	23/03/2012	Tristan Le Guern	Whole document	Import the TA2 documents
1	23/03/2012	Tristan Le Guern	1.2	Update information on libtuntap
2	23/03/2012	Tristan Le Guern	1.1	Update information on libtclt
3	23/03/2012	Nicolas Vivet	2.4 and 5	Enhance libevent and bugs parts
4	23/03/2012	Antoine Marandon	2.1	Update information relative to the logging API
5	23/03/2012	Tristan Le Guern	2.1	Unify the logging API
6	23/03/2012	Antoine Marandon	3	Add core part (pipeline + packet factory)
7	17/05/2012	Antoine Marandon	4	Add meta-connexion
8	18/05/2012	Nicolas Vivet	Abstract	Add an abstract
9	17/01/2013	Florent	libtclt	link to the libtclt documentation on phabricator

Contents

List of Figures

1 About this document

This document will teach you how to use our APIs and internals routines. We assume:

- That you already know C;
- That you already know network programming;
- That you are used to work with more than one OS;

2 Introduction

The tNETacle is a decentralized Virtual Private Networks (VPN) with security, ease of use and portability in mind.

Not to mention strong encryption, the tNETacle use the privilege separation. This means we set up everything that needs privilege, set up a socketpair() and fork(). The "large" process chroot() in a jail and revokes his privilege. It will be responsible of most of the work and sometimes ask to the "small" process, which retains is privilege, for specific operation.

The tNETacle works by default. There is no need to edit complicated files or to change command line switches: everything is done to work without help. A key concept is our user interface (UI): it is separated from the core daemon, unlike some others VPN. This is an important point since it allows the creation of well integrated or specific clients like command line tools (CLI), web interfaces (WebUI) or graphical application (GUI).

The tNETacle is meant to run everywhere in order to connect everybody, independently of there equipements. We are compatible with Windows NT, Mac OS X, Linux, FreeBSD, OpenBSD and NetBSD. These Operating Systems are our official targets, but there is no reason it doesn't work on others UNIX-like systems: the only requirements are a working C compiler and a tunnel driver (tun).

The code is organized in separated parts: independent libraries, the core engine and the official client.

3 Libraries

This is a list of libraries developped in conjunction with the tNETacle. They are exported separately to be reused by other projects.

3.1 Libtclt

The libtclt is described on the fabricator : http://trac.medu.se/w/projects/libtclt_api/.

3.1.1 Preliminaries

libtclt is written with simplicity and ease of utilisation in mind. It has to work with GUI and CLI clients on every Operating System supported by the tNETacle.

3.1.2 Setting up the library

Before every other call, you need to initialize the library with `tnt_tclt_init()`. It will set-up global variables and parameters.

Interface

```
void tnt_tclt_init(void);
```

3.1.3 Deallocating the library

If you have no more use of the library, when you shutdown the application for example, you should call `tnt_tclt_destroy()` in order to free the memory.

Interface

```
void tnt_tclt_destroy(void);
```

3.1.4 Getting the library version

In order to support updates without breaking up every application that uses it, the libtclt allows you to request its version number with `tnt_tclt_get_version()`.

It will return a numerical version number.

Interface

```
void tnt_tclt_get_version(void);
```

This library is still in an early stage of development.

3.2 Libtuntap

libtuntap is a library designed for configuring `TUN` and `TAP` devices in a portable manner. `TUN` and `TAP` are virtual networking devices which allow userland applications to receive packets sent to it. The userland applications can also send their own packets to the devices and they will be forwarded to the kernel.

We need these functionalities to build our private network.

3.2.1 Preliminaries

libtuntap is a library written with two goals in mind:

- Portability - A program written with the libtuntap should work accross all the platforms libtuntap support.
- Simplicity - The functionalities provided by the libtuntap are for configuration only. You will have to provide the functions to read and write from the devices.

3.2.2 Setting up a virtual device

The `tnt_tt_init()` function allocates and returns a new non-configured device. If there is an error, it returns `NULL`.

Interface

```
struct device *tnt_tt_init(void);
```

3.2.3 Deallocating virtual device

When you are finished with a device you can deallocate it with `tnt_tt_destroy()` or `tnt_tt_release()`. They will both release the memory, but the first one will also remove the device id supported by the Operating System.

Interface

```
void tnt_tt_destroy(struct device *);
void tnt_tt_release(struct device *);
```

3.2.4 Setting the device

After creating a device with `tnt_tt_init()`, you should configure it with `tnt_tt_start()`. This function will give the basic default to your device.

The second parameter corresponds to the OSI layer of your device, layer 2 or layer 3, respectively `TNT_TUNMODE_ETHERNET` and `TNT_TUNMODE_TUNNEL`.

The third parameter corresponds to the number of the device, if supported by the Operating System. For exrample, on an Unix, you can request specificaly `tun42` with that option. If you don't need a specific number, use `TNT_TUNID_ANY`.

If there is an error, `tnt_tt_start()` returns `-1`

Interface

```
#define TNT_TUNMODE_ETHERNET 1
#define TNT_TUNMODE_TUNNEL 2

int tnt_tt_start(struct device *, int, int);
```

3.2.5 Setting the MTU

libtuntap supports modifying the Maximum Transmission Unit of a virtual device at any time. There is no restriction on the value you can pass to `tnt_tt_set_mtu()`. If there is an error, `tnt_tt_set_mtu()` returns `-1`, but `tnt_tt_get_mtu()` never fails.

Interface

```
int tnt_tt_get_mtu(struct device *);
int tnt_tt_set_mtu(struct device *, int);
```

3.2.6 Setting the MAC Address

libtuntap supports modifying the Media Access Control of a virtual device. The second argument of `tnt_tt_set_hwaddr()` can be any valid MAC address or the string `"random"`. If there is an error, `tnt_tt_set_hwaddr()` returns `-1`, but `tnt_tt_get_hwaddr()` never fails.

Interface

```
char *tnt_tt_get_hwaddr(struct device *);
int tnt_tt_set_hwaddr(struct device *, const char *);
```

3.2.7 Setting the IP Address

`tnt_tt_set_ip()` handles both IPv6 and IPv4 format in unified interface. The first parameter is a valid IP address, the second a valid netmask, in the CIDR notation. Multiple calls to this function can be done, but only the last one will be taken into account. If there is an error, it returns `-1`.

Interface

```
int tnt_tt_set_ip(struct device *, const char *, const char *);
```

3.2.8 Bringing the device up

When all your configuration process is done, you can bring the device up (connect the virtual wire on Windows) with `tnt_tt_up()`.

The device will be ready to use.

Interface

```
int tnt_tt_up(struct device *);
```

3.2.9 Bringing the device down

You can bring the device down (disconnect the virtual wire on Windows) with `tnt_tt_down()`.

The configuration will not be affected but you will not receive any packets.

Interface

```
int tnt_tt_down(struct device *);
```

3.2.10 Exemple

This is an exemple of the use of libtuntap, under the ISC licence.

```
#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>
#include <arpa/inet.h>

#include <string.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <getopt.h>
#include "tun.h"

void
usage(void) {
        fprintf(stderr, "usage: -a addr/netmask -i interface -h");
        exit(1);
}

int
main(int argc, char *argv[]) {
        struct device *dev;
        int ch;
        char *addr;
        char *netmask;
        char *ptr;
        int interface;
        struct timeval tv;

        interface = TNT_TUNID_ANY;
        addr = "1.2.3.4";
        netmask = "255.255.255.0";
        while ((ch = getopt(argc, argv, "a:i:h:")) != -1) {
                switch (ch) {
                case 'a':
                        ptr = strchr(optarg, '/');
                        if (ptr == NULL)
                            usage();
                        *ptr = '\0';
                        ptr++;
                        addr = strdup(optarg);
                        netmask = strdup(ptr);
                        break;
                case 'i':
                        interface = atoi(optarg);
                        break;
                case 'h':
                default:
                        usage();
                }
        }

        dev = tnt_tt_init();
        if (tnt_tt_start(dev, TNT_TUNMODE_ETHERNET, interface) == -1)
            fprintf(stderr, "Can't start the device\n");
        printf("Interface: %s\n", tnt_tt_get_ifname(dev));

        if (tnt_tt_set_ip(dev, addr, netmask) == -1)
            fprintf(stderr, "Can't set ip\n");

        if (tnt_tt_up(dev) == -1)
            fprintf(stderr, "Can't bring interface up\n");

        tv.tv_sec = 5;
        tv.tv_usec = 0;
        select(1, NULL, NULL, NULL, &tv);

        if (tnt_tt_down(dev) == -1)
            fprintf(stderr, "Can't bring interface down\n");
        tnt_tt_destroy(dev);
        return 0;
}
```

3.3 Containers

With the tNETacle, we developed a new container library in C, which had been called "calm containers" by one of our team member.

The main purpose of the calm containers are to be light, and close in performance to what we can find with other low level libraries, and of course to be easy to use. The API is somewhat close to the C++'s STL. So if you are used to std::vector, you will not be to much surprised.

The main difference between the calm containers and the other generic containers that we can find on the internet are:

- the API is not in upper case because
- it's not macros, therefore it is
- easily debugable because all the function and types actually exists, and there is not a single `void *` in the code.

3.3.1 vector

The vector are the most simple type of containers, it is a plain dynamic array of elements. Random and streamed access are quick, and it's memory efficient. The size of the meta-data of the vector is only 24 bytes on a 64 bits system and 12 bytes on a 32 bits system (two size_t and one pointer) which is small.

As we are talking about generic vector without `void *`, we need generate as much function as there is types. And this bring us to the problem of naming. The chosen solution in calm containers is to let the developer decide of a mangling prefix via the macro VECTOR_PREFIX and a type via the macro VECTOR_TYPE.

```
#define VECTOR_TYPE foo
#define VECTOR_PREFIX bar
#include "vector.h"
#undef VECTOR_TYPE
#undef VECTOR_PREFIX
```

Don't forget to unset the defined macro value, otherwise the compiler will complain when you will use `vector.h` several times in a row.

All the following function will be generated, with the matching type and prefix.

```
struct vector_bar {
  foo *vec; /*the pointer to the allocated memory*/
  size_t size; /*the size of the vector in element*/
  size_t alloc_size; /*you'd better not touch this value*/
};

static inline void v_bar_init(struct vector_bar *v);
static inline void v_bar_push(struct vector_bar *v, foo *val);
static inline int v_bar_resize(struct vector_bar *v, size_t);
static inline void v_bar_insert_range(struct vector_bar *, foo *,
                                      foo *, foo *);
static inline void v_bar_insert(struct vector_bar *, foo *, foo *);
static inline void v_bar_pop(struct vector_bar *v);
static inline void v_bar_delete(struct vector_bar *v);
static inline foo *v_bar_begin(struct vector_bar *v);
static inline foo *v_bar_end(struct vector_bar *v);
static inline foo *v_bar_next(foo *it);
static inline void v_bar_erase(struct vector_bar *v, foo *ptr);
static inline void v_bar_erase_range(struct vector_bar *v, foo *,
                                     foo*);
static inline foo v_bar_at(struct vector_bar *v, size_t i);
static inline foo *v_bar_atref(struct vector_bar *v, size_t i);
static inline foo v_bar_front(struct vector_bar *v);
static inline foo v_bar_back(struct vector_bar *v);
static inline foo *v_bar_frontref(struct vector_bar *v);
static inline foo *v_bar_backref(struct vector_bar *v);
static inline foo *v_bar_find_if(struct vector_bar *v, foo *val,
                                 int (*)(foo const *, foo const *));
```

Note that this will be the only macro you will ever see, from now, all of those function are real function and exists in you binary. Notice also the absence of `void *`. Those functions are strongly typed, allowing the compiler to say pretty warning if you do something bad, and allowing the optimizer to understand what is going on.

The vectors works pretty much like an explicit instantiation of a C++ template. And the macro stand for the template arguments.

The usage of the vector is pretty close to the C++'s std::vector:

```
void
foobar(vector_bar *v)
{
  foo* it = v_bar_begin(v);
  foo* ite = v_bar_end(v);

  for (; it != ite; it = v_bar_next(it))
  {
    /*...*/
  }
}
```

```
  void
  foobar(vector_bar *v)
  {
    foo f;

    /* do something smart*/

    v_bar_push(v, &f); /* The f will be copied inside the vector, we are just
    using a pointer to avoid a copy*/
  }
```

4 Modules

Modules are small pieces of code that are not exported outside of the tNETacle. Their goal is to encapsulate external dependencies, like `zlib`, or to provide usefull routines.

4.1 Compression - zlib

4.1.1 Dependency

We use the `zlib` library, which is designed to be a free, general-purpose, legally unencumbered, lossless data-compression library for use on virtually any computer hardware and operating system.

It is encapsulated in a little module with a really simple interface, in "src/compress.c".

4.1.2 Compress

You can compress any data using the `tnt_compress()` function. It takes an `uchar *` (a string of **`unsigned char`**), its associated size, and the output size of the compressed string which will be set during the call. The function will then pass it to the Zlib, which will compress it and returns the compressed version. If an error occurs, it will return `NULL`.

Interface

```
typedef unsigned char uchar;

uchar *tnt_compress(uchar *, const size_t, size_t *);
```

4.1.3 Uncompress

You can uncompress any data using the `tnt_uncompress()` function. It takes the string to uncompress, its size and the original size of the string. If an error occurs, it will return `NULL`.

Interface

```
uchar *tnt_uncompress(uchar *, const size_t, const size_t);
```

4.2 Logging

The logging utility is able to report informational, debug, notice, warning and error messages. On UNIX-like systems they are sent to the system syslog utility; on Windows NT systems, they are written into a special file. In debug mode, however, they will be written on the standart output.

4.2.1 Setting up the API

Before every other call, you need to initialize the library with `log_init()`. It will set-up global variables, open log files or syslog utility.

Interface

```
void log_init(void);
```

4.2.2 Setting the prefix

In order to distinguish messages written from different processus, the logging API provide a way to set a prefix who will be printed before every messages. This is totally optional.

Interface

```
void log_set_prefix(char *);
```

4.2.3 Writing a message

To write a message in the logging sub-system, you only have to choose wicht type of message you want: Error with the system error message, simple Error, Warning with the system error message, Notice (not a warning but something that should probably be addressed), Informational and Debug.

The Error class messages take an integer as first parameter which will be passed to `exit()`.

```
void log_err(int, const char *, ...);
void log_errx(int, const char *, ...);
void log_warn(const char *, ...);
void log_warnx(const char *, ...);
void log_notice(const char *, ...);
void log_info(const char *, ...);
void log_debug(const char *, ...);
```

4.3 Encryption - OpenSSL

4.3.1 Dependency

For the encryption, we are using the library `OpenSSL`.

OpenSSL is an open source implementation of the SSL and TLS protocols.

This abstraction is placed in a module rather than in a library mainly because their is no other implementation that we can use:

- PolarSSL: Released under the non-free GPLv2 licence.
- CyaSSL: Released under the non-free GPLv2 licence and developped inside USA, which forbid exportation of cryptographic system.
- TropicSSL: A fork of PolarSSL released under the BSD licence, but the project is still young and untested.

Their is no API to show yet, it's still under development.

4.4 Events - libevent

4.4.1 Dependency

To handle system events (Network, IPC, ...), we use the library `Libevent`, famous for being powerful and stable. Many free and commercial projects already use it.

We have decided to make use of this library not only for these network facilities but also for the utilities functions it offers to the developers. In fact, the main advantage of the `libevent` is its portability and so we decided to make the most of it.

Their is no API to show yet, it's still under development, but it will handle both `libevent` version 1 and 2.

Documentation is available on the project website.

5 Core

5.1 Description

In practice, the core is the process running in the background of UNIX-like systems or the application hidden in the side-bar of Windows NT systems.

The core allows communication to other cores and can be set with a network socket. Each user need to have a core which contains all the logical part of the system to ensure a decentralized VPN.

Each core will be master of its own network and will have different accessibility policy relative to the other nodes. It is a peer-to-peer application.

5.2 Architecture

5.2.1 Privilege Separation

For an enhanced security, the tNETacle use the privilege separation. That means we have two separated proccess: one with privilege and one without. The first one does only little jobs like opening the tunnel device while the second one does all the logic, as shown in figure 1. The goal of this architecture is to avoid running too much code in an administrator privileged process as well as reducing the attack surface of the tNETacle: a bug or an exploit will only impact on the process with no privilege.

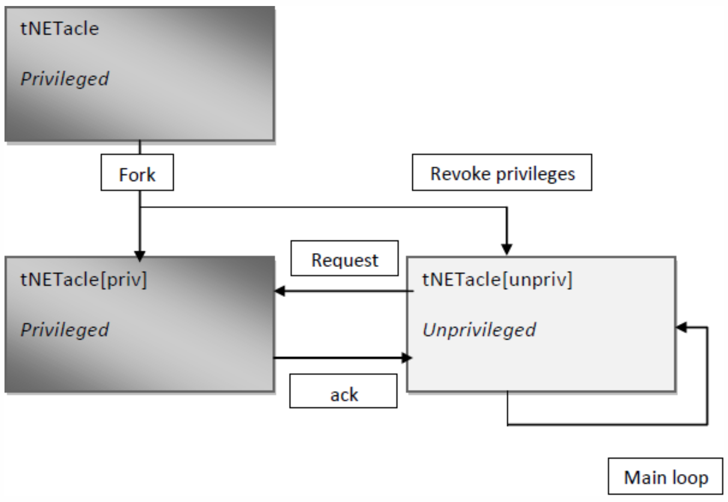

Figure 1: Privilege separation in the tNETacle.

The communication between these two process is handle via the imsg framework, from the OpenBSD project. It allows the creation and dispatching of messages which contain a general header and a variable (most of the time a file descriptor).

These headers are actually: *Interface*

```
enum imsg_type {
        IMSG_NONE,
        IMSG_CREATE_DEV,
        IMSG_SET_IP
};
```

`IMSG_NONE` is actualy unused, it's a safeguard for debugging purpose. `IMSG_CREATE_DEV` is a request to create a particular tunnel device. `IMSG_SET_IP` allow a configuration of the IP address of a tunnel device.

The way they are sent will not be shown here, because there are really seldom used. However, if you are still interested, you should read these files:

- include/imsg.h
- include/tnetacle.h
- sys/unix/tnetacle.c
- sys/unix/tnetacled.c

5.2.2 Pipeline

A pipeline has been created to handle the different process that a packet has to go through in tNETacle. The pipeline is differenced from downstream and upstream, and can be used to create or decode packets. Notable process that might occur through the pipeline are compression and encryption.

Each modules of the pipeline are registred and called by:

`int modulename_upstream(t_message)` or `modulename_downstream(t_message)`. t_message contains the raw data and usefull informations about the sender and connection (like public key).

Each message contain informations about which module should be called to decode itself. For more information, please refere to pipeline.h

5.2.3 Packet Factory

At the top of the downstream pipeline, and at the bottom of the upstream pipeline, the packet factory is used. The main goal of the packet factory is to setup or decode the informations that are needed by the data to navigate through the pipeline.

5.2.4 Meta-connexion

General description

The meta-connexion (MC) is a reliable and secure link that allow communication of meta-informations from a core to an other. Theses informations include public key research, authentication, and other. Theses connexions do NOT transit data generated by tiers software. Usage of bandwich is as limited as possible to prevent excessive overhead. Due to low bandwich consumption and required realiability, the meta-connexions are established in TCP/IP.
From here, the term MC refer to the **`struct`** `mc` and the implementation of the meta connexion in the tNETacle.

Technical description

MC are still under construction

MC is an abstraction over the bufferevents. Their goal is to handle different corner cases and to asserts that internals and connexions are established correctly. Please read the libevent documentation about the bufferevents for overall knowledge.

Structures

The data representing a MC is described in a **`struct`** `mc`. Defined in `include/mc.h`.

```
struct mc
{
 struct peer {
   struct sockaddr *address;
   int len;
 } p;
 struct bufferevent *bev;
};
```

The current MCs are stocked in a plain vector, in a **`struct`** `vector` Defined in `include/server.h`.

```
struct server {
 /*...*/
 struct vector_mc peers;
 /*...*/
};
```

The **`struct`** `vector_mc` is a type defined by the header `vector.h`. See documentation page 9.

Functions

`include/mc.h`

```
void mc_init(struct mc *, struct sockaddr *, int len, struct bufferevent *bev);
void mc_close(struct mc *);
```

Functions `mc_init()` and `mc_close()` can be used to init or to close a meta-connexion. Those two functions does not allocate nor free memory. They are called to construct and to destroy the **`struct`** `mc`.

For `mc_init` the parameter **`struct`** `bufferevent *`, must be a correctly allocated bufferevent.

6 Client

6.1 Description

The client is used to configure the tNETacle-core. It is a wrapper between the user and the system. It communicates with the tNETacle-core thanks to a network connection.

6.2 Langage

The client will be written in C++ with the Qt library.

6.3 Architecture

The design is not fixed yet.

7 Bugs

7.1 Contributions

Please report any bugs by filling in a ticket on our dedicated interface http://trac.medu.se. The ticket must be as precise as possible and include which version was used and the context of utilisation. If your ticket is accepted, it will be assigned to a developer who will be responsible to for providing a solution. You can also provide a solution to a bug by attaching a patch to your ticket. This patch will be reviewed by the core developers and merged to the project if the code respects the rules described on the project website.

7.2 Known issues

The known issues are summurized on the following table:

Number	Where	Description	Solved
01	libtuntap	Cannot create the interface	Yes
02	libtuntap	Incorrect behavior when setting a MAC Address	Yes
03	libtuntap	Can't assign a MAC Address under OpenBSD	Yes
04	libtuntap	Can't assign a MTU without blowing up the device	Yes
05	libtuntap	Can't assign an IPv6	No