Querying libiptc HOWTO

Leonardo Balliache

leonardo@opalsoft.net

Version 0.1 − April 30, 2002

Revision History		
Revision 0.1	2002−04−30	Revised by: lb
Initial release.		

Table of Contents

Table of Contents

1. Legal Notice

2. Translations

If you want to translate this document you are free to do so. However, you will need to do the following:

1. Check first that another version of the document doesn't already exist at your local LDP.
2. Maintain all sections (including 'Legal Notice', 'Translation', 'Credits', etc., etc.) of the document.
3. No need to ask me to translate! You just have to let me know (if you want) about your translation.

Thank for your translation.

3. Disclaimer

I took this "disclaimer" from Linux−Advance Networking Overview by Saravanan Radhakrishnan (08−1999) because it applies in my own case:

All the text in this document is completely based on my understanding of implementations of various features. I have read some documents and have seen the code myself, and I described them based on my understanding. If the reader notices any concept description which appears to be contrary to their understanding of the concept, the issue can be taken up for discussion and corrections will be made to the document as necessary. I would appreciate any suggestions and comments made in order to improve the quality of this document.

4. Credits

I want to thank the following people and organizations who had helped me, directly or not, to make this document possible:

- My wife Cielo and my sons Jose, Dario and Gustavo by their patient and support.
- Linux Documentation Project for publishing and uploading my document.
- The site http://www.docum.org drived by Stef Coene that give me some ideas for writing this document.
- Paul "Rusty" Rusell who write the kernel firewall code, the excelent package *iptables* and the associated library *libiptc*.
- Harald Welte who write the utility *iptables−save*.
- Alexey Kuznetsov who write the kernel queue discipline code and the excelent package *iproute2*.
- Tabatha Persad from the Linux Documentation Project who revised my english syntax and writing, gave me several ideas to improve the content and encouraged me to learn and use DocBook to write the final version of this document.

---

5. Objectives

This HOWTO explains how to use the *libiptc* library included in the *iptables* package. This document can show you how to use short C or C++ programs to query the internal structure of the firewalling code, to check chains and rules, packet and byte counters, and in a second phase, if you are a little "brave", to modify them.

You can find the latest version of this document at Querying libiptc HOWTO.html.

If you have suggestions to help make this document better, please submit your ideas to me at the following address: leonardo@opalsoft.net.

While I wrote this HOWTO, I developed a simple bandwith meter using user−defined chains to get the data to be measured. This idea was conceived looking at **monitor.pl**, a simple perl program for bandwith measurement, written by Stef Coene at http://www.docum.org. I recommend this site to people interested in bandwidth control and measurement.

6. What is libiptc?

libiptc is the library that is used to communicate with netfilter, the internal kernel code in charge of firewalling and packet filtering. This code and *iptables* were written by Paul "Rusty" Russell. *iptables* was developed using *libiptc* calls to get the job done.

If you want to have more information about *iptables*, *libiptc* and the firewalling code, have a look at links at the end of this document.

7. How did I obtain this knowledge?

Just looking at code in *iptables 1.2.6* package and especially at program *iptables−save.c* that use *libiptc* to dump information from firewalling kernel code.

I will try to be very pragmatic and clear in order to make this HOWTO useful.

8. Previous knowledge and system requirements

You have to have some previous knowledge to follow this document:

1. *Very important:* You must know how to use the *iptables* package as a user, such as how to create or list rules and user chains. You do not need to be a firewall expert, but you should know how to use *iptables* fluently.
2. You have to have kernel sources installed in your system, in `/usr/src/linux` as usual.

   I am using a *2.4.16* kernel in a *SuSE 7.1* Linux environment. You need *2.4.x* kernel code to follow this HOWTO, preferably kernel *2.4.16*. For *SuSE* you can get the kernel sources at ftp://ftp.gwdg.de/pub/linux/suse/ftp.suse.com/suse/i386/update.
3. You have to know how to compile the kernel if you have to update your kernel version. After activating the netfilter options using **make menuconfig**, you must compile and install the kernel as usual.
4. Reboot your new kernel using **init 6**. Ensure that you backup a copy of your previous kernel in *lilo* in case you encounter a problem and need to retrace your steps.
5. Be sure that your new *2.4.x* kernel is running fine. To install *iptables−1.2.6* you will need to patch the kernel again (and re−compile and install it), and it is better if you follow the previous two steps to ensure that your kernel is running right before applying new iptables patches.

9. Installing iptables + libiptc

To install *libiptc* follow these steps:

1. Download *iptables−1.2.6.tar.bz2* from http://netfilter.samba.org/.
2. Copy the *iptables* tar file into `/usr/local/src`:
   ```
   bash# cp iptables-1.2.6.tar.bz2 /usr/local/src
   ```
3. Unpack:
   ```
   bash# tar xjvf iptables-1.2.6.tar.bz2
   ```
4. Go into the iptables directory:
   ```
   bash# cd iptables-1.2.6
   ```
5. Check to see if your kernel needs some aditional patches with:
   ```
   bash# make pending-patches KERNEL_DIR=/usr/src/linux
   ```

   If your kernel source is located somewhere other than in `/usr/src/linux`, replace the kernel source directory in the command line above with your source directory.

   Be careful with this option. This command invokes *patch−o−matic*, a new patch verification utility by Rusty Russell. The utility will show you a list of new patches (some proposed, some submitted, some accepted) available for your kernel source. As Rusty himself says, "Some of these new patches have bugs", and you do not have to apply all of them.

   Read the information showed for each patch carefully and answer with **y** (apply the patch) or **N** (skip this patch). In some cases answering **y** will try to apply the patch, but if the patch finds some differences between your sources, it will be skipped and the next new one presented.

   I did not apply any of the proposed patches and kept my kernel in its original state before continuing to the next step.
6. Make the iptables package with:
   ```
   bash# make KERNEL_DIR=/usr/src/linux
   ```

   Again, if your kernel source is not at `/usr/src/linux`, replace the kernel source directory in the command above.

   If all goes right the compiler will finish without errors.
7. Before the next step, check to see if you have installed iptables package by typing:
   ```
   bash# rpm -q iptables
   ```

   If the iptables rpm is installed, you will see the name and version of the package, similar to:

   iptables−1.1.2−13

   In this case un−install with:

   ```
   bash# rpm -e iptables
   ```
8. Install the new created package:
   ```
   bash# make install KERNEL_DIR=/usr/src/linux
   ```

Again, check your kernel source directory.

This command will install the binaries (*iptables, iptables−save, iptables−restore*) in `/usr/local/sbin`, the manuals in `/usr/local/man/man8` and the modules in `/usr/local/lib/iptables`.

9. Finally install the headers, development libraries and associated development man pages, with:

```
bash# make install-devel
```

This command will install the *libiptc* library in `/usr/local/lib`.

I think something must be wrong with this command. It does not install all headers files properly, so you must install them yourself using:

```
bash# cd /usr/local/src/iptables-1.2.6
bash# cp include/iptables.h /usr/local/include
bash# cp include/iptables_common.h /usr/local/include
bash# mkdir /usr/local/include/libiptc
bash# cp include/libiptc/libiptc.h /usr/local/include/libiptc
bash# cp include/libiptc/ipt_kernel_headers.h /usr/local/include/libiptc
bash# cp iptables.o /usr/local/lib
```

`iptables.o` is needed above to compile programs to get rule information from netfilter.

Now you are ready to create programs that can communicate directly with libiptc.

10. How to create your program(s)

Create your program(s) in `/usr/local/src`; this way you will not have problems with gcc looking for files in the "include" section.

Your program(s) would be something like this:

```
/* My program */

#include <getopt.h>
#include <sys/errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h>
#include <time.h>
#include "libiptc/libiptc.h"
#include "iptables.h"

int main(void)
{
  /* Use always this part for your programs .... From here ... **** */
  iptc_handle_t h;
  const char *chain = NULL;
  const char *tablename = NULL;

  program_name = "my_program";
  program_version = NETFILTER_VERSION;
  /* .... To here .... ******************************************** */

  /* From here you write your own code */
  .... your code ...
  ....
  ....

} /* main */
```

- The "include" section is *a must* in your c/c++ program(s).
- If you are using c++ do not forget to write extern "C" for these include.

11. Functions to query libiptc

This section explains which functions allow you to query libiptc. We will use the header file of *libiptc*, file `usr/local/include/libiptc/libiptc.h`, containing prototypes of each function as a reference to develop our explanation.

I have also included a brief description (when available) taken from Linux netfilter Hacking HOWTO within each function explanation.

11.1. iptc_init

Name: iptc_init

Usage: Takes a snapshot of the rules.

Prototype: iptc_handle_t iptc_init(const char *tablename)

Description: This function must be called as initiator before any other function can be called.

Parameters: *tablename* is the name of the table we need to query and/or modify; this could be *filter*, *mangle*, *nat*, etc.

Returns: Pointer to a structure of type *iptc_handle_t* that must be used as main parameter for the rest of functions we will call from *libiptc*. *iptc_init* returns the pointer to the structure or NULL if it fails. If this happens you can invoke *iptc_strerror* to get information about the error. See below.

Have a look at this section of code in file `iptables-save.c` for how to invoke this function:

```
h = iptc_init(tablename);
if (!h)
  exit_error(OTHER_PROBLEM, "Can't initialize: %s\n",iptc_strerror(errno));
```

11.2. iptc_strerror

Name: iptc_strerror

Usage: Translates error numbers into more human–readable form.

Prototype: const char *iptc_strerror(int err)

Description: This function returns a more meaningful explanation of a failure code in the iptc library. If a function fails, it will always set *errno*. This value can be passed to *iptc_strerror()* to yield an error message.

Parameters: *err* is an integer indicating the error number.

Returns: Char pointer containing the error description.

11.3. iptc_first_chain

Name: iptc_first_chain

Usage: Iterator functions to run through the chains.

Prototype: const char *iptc_first_chain(iptc_handle_t *handle)

Description: This function returns the first chain name in the table.

Parameters: Pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Char pointer to the name of the chain.

11.4. iptc_next_chain

Name: iptc_next_chain

Usage: Iterator functions to run through the chains.

Prototype: const char *iptc_next_chain(iptc_handle_t *handle)

Description: This function returns the next chain name in the table; NULL means no more chains.

Parameters: Pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Char pointer to the name of the chain.

These two previous functions allow to us to iterate through the chains of the table getting the name of each of the chains; *iptc_first_chain* returns the name of the first chain of the table; *iptc_next_chain* returns the name of next chains and NULL when the function reaches the end.

We can create *Program #1* to exercise our understanding of these previous four functions:

```
/*
 * How to use libiptc- program #1
 * /usr/local/src/p1.c
 */

#include <getopt.h>
#include <sys/errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h>
#include <time.h>
#include "libiptc/libiptc.h"
#include "iptables.h"

int main(void)
```

```
{
  iptc_handle_t h;
  const char *chain = NULL;
  const char *tablename = "filter";

  program_name = "p1";
  program_version = NETFILTER_VERSION;

  h = iptc_init(tablename);
  if ( !h )   {
     printf("Error initializing: %s\n", iptc_strerror(errno));
    exit(errno);
  }

  for (chain = iptc_first_chain(&h); chain; chain = iptc_next_chain(&h))  {
    printf("%s\n", chain);
  }

  exit(0);

} /* main */
```

Write this program and save it as `p1.c` in `/usr/local/src`.

Now write this "bash" script to simplify the compiling process:

```
#!/bin/bash

gcc -Wall -Wunused -DNETFILTER_VERSION=\"1.2.6\" -rdynamic -o $1 $1.c \
/usr/local/lib/iptables.o /usr/local/lib/libiptc.a -ldl
```

Save it as `ipt-cc` and do not forget to *chmod 0700 ipt−cc*.

Now compile your *p1* program:

```
bash# ./ipt-cc p1
```

And run it:

```
bash# ./p1
```

You will get:

```
INPUT
FORWARD
OUTPUT
```

These are the three built−in iptables chains.

Now create some new chains using iptables and run your program again:

```
bash# iptables -N chain_1
bash# iptables -N chain_2
bash# ./p1
```

You will get:

```
INPUT
FORWARD
OUTPUT
chain_1
chain_2
```

Try to generate an error initializing tablename to *myfilter* instead of *filter*. When you compile and execute your program again, you will get:

```
Error initializing: Table does not exist (do you need to insmod?)
```

iptables informs you that *myfilter* does not exist as a table.

11.5. iptc_is_chain

Name: iptc_is_chain

Usage: Check if a chain exists.

Prototype: int iptc_is_chain(const char *chain, const iptc_handle_t handle)

Description: This function checks to see if the chain described in the parameter *chain* exists in the table.

Parameters: *chain* is a char pointer containing the name of the chain we want to check to. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: integer value 1 (true) if the chain exists; integer value 0 (false) if the chain does not exist.

11.6. iptc_builtin

Name: iptc_builtin

Usage: Is this a built−in chain?

Prototype: int iptc_builtin(const char *chain, const iptc_handle_t handle)

Description: This function is used to check if a given chain name is a built−in chain or not.

Parameters: *chain* is a char pointer containing the name of the chain we want to check to. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if the given chain name is the name of a builtin chain; returns integer value 0 (false) is not.

11.7. iptc_first_rule

Name: iptc_first_rule

Usage: Get first rule in the given chain.

Prototype: const struct ipt_entry *iptc_first_rule(const char *chain, iptc_handle_t *handle)

Description: This function returns a pointer to the first rule in the given chain name; NULL for an empty chain.

Parameters: *chain* is a char pointer containing the name of the chain we want to get the rules to. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns a pointer to an *ipt_entry* structure containing information about the first rule of the chain. See below for an explanation of this structure.

11.8. iptc_next_rule

Name: iptc_next_rule

Usage: Get the next rule in the given chain.

Prototype: const struct ipt_entry *iptc_next_rule(const struct ipt_entry *prev, iptc_handle_t *handle)

Description: This function returns a pointer to the next rule in the given chain name; NULL means the end of the chain.

Parameters: *prev* is a pointer to a structure of type *ipt_entry* that must be obtained first by a previous call to the function *iptc_first_rule*. In order to get the second and subsequent rules you have to pass a pointer to the structure containing the information about the previous rule of the chain. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns a pointer to an *ipt_entry* structure containing information about the next rule of the chain. See below for an explanation of this structure.

11.9. iptc_get_target

Name: iptc_get_target

Usage: Get a pointer to the target name of this entry.

Prototype: const char *iptc_get_target(const struct ipt_entry *e, iptc_handle_t *handle)

Description: This function gets the target of the given rule. If it is an extended target, the name of that target is returned. If it is a jump to another chain, the name of that chain is returned. If it is a verdict (eg. DROP), that name is returned. If it has no target (an accounting−style rule), then the empty string is returned. Note that this function should be used instead of using the value of the *verdict* field of the *ipt_entry* structure directly, as it offers the above further interpretations of the standard verdict.

Parameters: *e* is a pointer to a structure of type *ipt_entry* that must be obtained first by a previous call to the function *iptc_first_rule* or the function *iptc_next_rule*. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns a char pointer to the target name. See *Description* above for more information.

Now it is time to explain the *ipt_entry* structure; these pieces of code are taken from *iptables* package sources:

```
/* Internet address. */
struct in_addr {
  __u32  s_addr;
};

/* Yes, Virginia, you have to zero the padding. */
struct ipt_ip {
  /* Source and destination IP addr */
  struct in_addr src, dst;
  /* Mask for src and dest IP addr */
  struct in_addr smsk, dmsk;
  char iniface[IFNAMSIZ], outiface[IFNAMSIZ];
  unsigned char iniface_mask[IFNAMSIZ], outiface_mask[IFNAMSIZ];

  /* Protocol, 0 = ANY */
  u_int16_t proto;

  /* Flags word */
  u_int8_t flags;
  /* Inverse flags */
  u_int8_t invflags;
};

struct ipt_counters
{
  u_int64_t pcnt, bcnt;      /* Packet and byte counters */
};

/* This structure defines each of the firewall rules.  Consists of 3
   parts which are 1) general IP header stuff 2) match specific
   stuff 3) the target to perform if the rule matches */
struct ipt_entry
{
  struct ipt_ip ip;

  /* Mark with fields that we care about. */
  unsigned int nfcache;

  /* Size of ipt_entry + matches */
  u_int16_t target_offset;
  /* Size of ipt_entry + matches + target */
  u_int16_t next_offset;

  /* Back pointer */
  unsigned int comefrom;

  /* Packet and byte counters. */
  struct ipt_counters counters;
```

```
  /* The matches (if any), then the target. */
  unsigned char elems[0];
};
```

An *ipt_entry* structure contains:

- An *ipt_ip* structure containing (for the rule) the source address and netmask *(ip.src.s_addr, ip.smsk.s_addr)*, the destination address and netmask *(ip.dst.s_addr, ip.dmsk.s_addr)*, the protocol *(ip.proto)*, a flags field *(invflags)* to check for inverse *(!)* selections *(eg. ! 192.168.2.0/24, ! eth0, ! tcp, etc)*, the input interface *(iniface)*, the output interface *(outiface)*, the input *(iniface_mask)* and output *(outiface_mask)* interface masks and the *flags* field to check for fragmented packets.
- An *ipt_counters* structure containing the packet *(pcnt)* and byte *(bcnt)* counter of the rule. This information is important for bandwidth measurement.
- *target_offset* that is used to get the target information of the rule.
- Unknown fields: *nfcache, comefrom, elems, next_offset*. If someone can give me a feedback about these fields I would be grateful.

A simple way to work with all this information is to borrow some functions from `iptables-save.c` by Paul Russell and Harald Welte.

Here is another sample program *Program #2* written with a lot of help from Russell−Welte:

```
/*
 * How to use libiptc- program #2
 * /usr/local/src/p1.c
 */

#include <getopt.h>
#include <sys/errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h>
#include <time.h>
#include "libiptc/libiptc.h"
#include "iptables.h"

/* Here begins some of the code taken from iptables-save.c **************** */
#define IP_PARTS_NATIVE(n)       \
(unsigned int)((n)>>24)&0xFF,    \
(unsigned int)((n)>>16)&0xFF,    \
(unsigned int)((n)>>8)&0xFF,     \
(unsigned int)((n)&0xFF)

#define IP_PARTS(n) IP_PARTS_NATIVE(ntohl(n))

/* This assumes that mask is contiguous, and byte-bounded. */
static void
print_iface(char letter, const char *iface, const unsigned char *mask,
      int invert)
{
  unsigned int i;

  if (mask[0] == 0)
    return;
```

```
  printf("-%c %s", letter, invert ? "! " : "");

  for (i = 0; i < IFNAMSIZ; i++) {
    if (mask[i] != 0) {
      if (iface[i] != '\0')
        printf("%c", iface[i]);
    } else {
      /* we can access iface[i-1] here, because
       * a few lines above we make sure that mask[0] != 0 */
      if (iface[i-1] != '\0')
        printf("+");
      break;
    }
  }

  printf(" ");
}

/* These are hardcoded backups in iptables.c, so they are safe */
struct pprot {
  char *name;
  u_int8_t num;
};

/* FIXME: why don't we use /etc/protocols ? */
static const struct pprot chain_protos[] = {
  { "tcp", IPPROTO_TCP },
  { "udp", IPPROTO_UDP },
  { "icmp", IPPROTO_ICMP },
  { "esp", IPPROTO_ESP },
  { "ah", IPPROTO_AH },
};

static void print_proto(u_int16_t proto, int invert)
{
  if (proto) {
    unsigned int i;
    const char *invertstr = invert ? "! " : "";

    for (i = 0; i < sizeof(chain_protos)/sizeof(struct pprot); i++)
      if (chain_protos[i].num == proto) {
        printf("-p %s%s ",
               invertstr, chain_protos[i].name);
        return;
      }

    printf("-p %s%u ", invertstr, proto);
  }
}

static int print_match(const struct ipt_entry_match *e,
      const struct ipt_ip *ip)
{
  struct iptables_match *match
    = find_match(e->u.user.name, TRY_LOAD);

  if (match) {
    printf("-m %s ", e->u.user.name);

    /* some matches don't provide a save function */
    if (match->save)
```

```
      match->save(ip, e);
  } else {
    if (e->u.match_size) {
      fprintf(stderr,
        "Can't find library for match `%s'\n",
        e->u.user.name);
      exit(1);
    }
  }
  return 0;
}

/* print a given ip including mask if neccessary */
static void print_ip(char *prefix, u_int32_t ip, u_int32_t mask, int invert)
{
  if (!mask && !ip)
    return;

  printf("%s %s%u.%u.%u.%u",
    prefix,
    invert ? "! " : "",
    IP_PARTS(ip));

  if (mask != 0xffffffff)
    printf("/%u.%u.%u.%u ", IP_PARTS(mask));
  else
    printf(" ");
}

/* We want this to be readable, so only print out neccessary fields.
 * Because that's the kind of world I want to live in.  */
static void print_rule(const struct ipt_entry *e,
    iptc_handle_t *h, const char *chain, int counters)
{
  struct ipt_entry_target *t;
  const char *target_name;

  /* print counters */
  if (counters)
    printf("[%llu:%llu] ", e->counters.pcnt, e->counters.bcnt);

  /* print chain name */
  printf("-A %s ", chain);

  /* Print IP part. */
  print_ip("-s", e->ip.src.s_addr,e->ip.smsk.s_addr,
      e->ip.invflags & IPT_INV_SRCIP);

  print_ip("-d", e->ip.dst.s_addr, e->ip.dmsk.s_addr,
      e->ip.invflags & IPT_INV_DSTIP);

  print_iface('i', e->ip.iniface, e->ip.iniface_mask,
        e->ip.invflags & IPT_INV_VIA_IN);

  print_iface('o', e->ip.outiface, e->ip.outiface_mask,
        e->ip.invflags & IPT_INV_VIA_OUT);

  print_proto(e->ip.proto, e->ip.invflags & IPT_INV_PROTO);

  if (e->ip.flags & IPT_F_FRAG)
    printf("%s-f ",
           e->ip.invflags & IPT_INV_FRAG ? "! " : "");
```

```
  /* Print matchinfo part */
  if (e->target_offset) {
    IPT_MATCH_ITERATE(e, print_match, &e->ip);
  }

  /* Print target name */
  target_name = iptc_get_target(e, h);
  if (target_name && (*target_name != '\0'))
    printf("-j %s ", target_name);

  /* Print targinfo part */
  t = ipt_get_target((struct ipt_entry *)e);
  if (t->u.user.name[0]) {
    struct iptables_target *target
      = find_target(t->u.user.name, TRY_LOAD);

    if (!target) {
      fprintf(stderr, "Can't find library for target `%s'\n",
        t->u.user.name);
      exit(1);
    }

    if (target->save)
      target->save(&e->ip, t);
    else {
      /* If the target size is greater than ipt_entry_target
       * there is something to be saved, we just don't know
       * how to print it */
      if (t->u.target_size !=
          sizeof(struct ipt_entry_target)) {
        fprintf(stderr, "Target `%s' is missing "
            "save function\n",
          t->u.user.name);
        exit(1);
      }
    }
  }
  printf("\n");
}
/* Here ends some of the code taken from iptables-save.c ***************** */

int main(void)
{
  iptc_handle_t h;
  const struct ipt_entry *e;
  const char *chain = NULL;
  const char *tablename = "filter";
  const int counters = 1;

  program_name = "p2";
  program_version = NETFILTER_VERSION;

  /* initialize */
  h = iptc_init(tablename);
  if ( !h )   {
     printf("Error initializing: %s\n", iptc_strerror(errno));
    exit(errno);
  }

  /* print chains and their rules */
  for (chain = iptc_first_chain(&h); chain; chain = iptc_next_chain(&h))  {
```

```
    printf("%s\n", chain);
    for (e = iptc_first_rule(chain, &h); e; e = iptc_next_rule(e, &h))  {
      print_rule(e, &h, chain, counters);
    }
  }

  exit(0);

} /* main */
```

The function *print_rule* borrowed from `iptables-save.c` prints the information about a rule into a readable form using:

- *print_ip* to print the addresses,
- *print_iface* to print the interfaces,
- *print_proto* to print the protocols,
- *iptc_get_target* to get and print the targets (using *save*).

In *main* we iterate through each chain and for each one we iterate through each rule printing it.

The arguments of *print_rule* are:

- e = pointer to an *ipt_entry* structure containing information about the rule.
- h = pointer to an *iptc_handle_t* structure returned by *iptc_init*.
- chain = name of the chain.
- counters = 0: do not print counters; 1: print them.

OK, compile and run program *p2*:

```
bash# ./ipt-cc p2
bash# ./p2
```

You will get:

```
INPUT
FORWARD
OUTPUT
chain_1
chain_2
```

Now modify the environment using *iptables* to add some rules:

```
bash# iptables -A INPUT -p tcp -i eth0 -s ! 192.168.1.1 --dport 20 -j ACCEPT
bash# iptables -A chain_1 -p udp -o eth1 -s 192.168.2.0/24 --sport 33 -j DROP
```

Now if you run again *p2* you will get:

```
INPUT
[0:0] -A INPUT -s ! 192.168.1.1 -i eth0 -p tcp -m tcp --dport 20 -j ACCEPT
FORWARD
OUTPUT
chain_1
[0:0] -A chain_1 -s 192.168.2.0/255.255.255.0 -o eth1 -p udp -m udp --sport 33 -j DROP
chain_2
```

We have now rules printed for *INPUT* and *chain_1* chains. The numbers in the brackets at left are packet and byte counters respectively.

11.10. iptc_get_policy

Name: iptc_get_policy

Usage: Get the policy of a given built−in chain.

Prototype: const char *iptc_get_policy(const char *chain, struct ipt_counters *counter, iptc_handle_t *handle)

Description: This function gets the policy of a built−in chain, and fills in the *counters* argument with the hit statistics on that policy.

Parameters: You have to pass as arguments the name of the built−in chain you want to get the policy to, a pointer to an *ipt_counters* structure to be filled by the function and the *iptc_handle_t* structure identifying the table we are working to. The *ipt_counters* structure was explained in previous section; do not forget that *iptc_handle_t* must be obtained by a previous call to the function *iptc_init*.

Returns: Returns a char pointer to the policy name.

Using pieces of programs 1 and 2 we can write *program #3*:

```
/*
 * How to use libiptc- program #3
 * /usr/local/src/p3.c
 */

#include <getopt.h>
#include <sys/errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h>
#include <time.h>
#include "libiptc/libiptc.h"
#include "iptables.h"

int main(void)
{
  iptc_handle_t h;
  const char *chain = NULL;
  const char *policy = NULL;
  const char *tablename = "filter";
  struct ipt_counters counters;

  program_name = "p3";
  program_version = NETFILTER_VERSION;

  /* initialize */
  h = iptc_init(tablename);
  if ( !h )   {
```

```
    printf("Error initializing: %s\n", iptc_strerror(errno));
    exit(errno);
  }

  /* print built-in chains, their policies and counters */
  printf("BUILT-IN   POLICY  PKTS-BYTES\n");
  printf("-----------------------------\n");
  for (chain = iptc_first_chain(&h); chain; chain = iptc_next_chain(&h))  {
    if ( !iptc_builtin(chain, h) )
      continue;
    if ( (policy = iptc_get_policy(chain, &counters, &h)) )
      printf("%-10s %-10s [%llu:%llu]\n",
             chain, policy, counters.pcnt, counters.bcnt);
  }

  exit(0);

} /* main */
```

OK, compile and run program *p3*:

```
bash# ./ipt-cc p3
bash# ./p3
```

You will get something like this:

```
BUILT-IN  POLICY  PKTS-BYTES
----------------------------
INPUT     ACCEPT     [0:0]
FORWARD   ACCEPT     [0:0]
OUTPUT    ACCEPT     [0:0]
```

11.11. iptc_read_counter

Name: iptc_read_counter

Usage: Read counters of a rule in a chain.

Prototype: struct ipt_counters *iptc_read_counter(const ipt_chainlabel chain, unsigned int rulenum, iptc_handle_t *handle);

Description: This function read and returns packet and byte counters of the entry rule in chain *chain* positioned at *rulenum*. Counters are returned in a pointer to a type structure *ipt_counters*. Rule numbers start at 1 for the first rule.

Parameters: *chain* is a char pointer to the name of the chain to be readed; *rulenum* is an integer value defined the position in the chain of rules of the rule which counters will be read. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns a pointer to an *ipt_counters* structure containing the byte and packet counters readed.

12. Functions to modify firewalling rules and statistics

For those of you who are a little brave, *libiptc* has a group of functions to directly modify the firewalling rules and statistics *(use of iptables is really the safest way).*

These functions are not covered by this HOWTO and I will limit myself to presenting improved information taken from `libiptc.h` and the Linux netfilter Hacking HOWTO by Rusty Russell.

12.1. iptc_commit

Name: iptc_commit

Usage: Makes the actual changes.

Prototype: int iptc_commit(iptc_handle_t *handle)

Description: The tables that you change are not written back until the *iptc_commit()* function is called. This means it is possible for two library users operating on the same chain to race each other; locking would be required to prevent this, and it is not currently done. There is no race with counters, however; counters are added back in to the kernel in such a way that counter increments between the reading and writing of the table still show up in the new table. *To protect the status of the system you must commit your changes.*

Parameters: *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.2. iptc_insert_entry

Name: iptc_insert_entry

Usage: Insert a new rule in a chain.

Prototype: int iptc_insert_entry(const ipt_chainlabel chain, const struct ipt_entry *e, unsigned int rulenum, iptc_handle_t *handle)

Description: This function insert a rule defined in structure type *ipt_entry* in chain *chain* into position defined by integer value *rulenum*. Rule numbers start at 1 for the first rule.

Parameters: *chain* is a char pointer to the name of the chain to be modified; *e* is a pointer to a structure of type *ipt_entry* that contains information about the rule to be inserted. The programmer must fill the fields of this structure with values required to define his or her rule before passing the pointer as parameter to the function. *rulenum* is an integer value defined the position in the chain of rules where the new rule will be inserted. Rule numbers start at 1 for the first rule. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.3. iptc_replace_entry

Name: iptc_replace_entry

Usage: Replace an old rule in a chain with a new one.

Prototype: int iptc_replace_entry(const ipt_chainlabel chain, const struct ipt_entry *e, unsigned int rulenum, iptc_handle_t *handle)

Description: This function replace the entry rule in chain *chain* positioned at *rulenum* with the rule defined in structure type *ipt_entry*. Rule numbers start at 1 for the first rule.

Parameters: *chain* is a char pointer to the name of the chain to be modified; *e* is a pointer to a structure of type *ipt_entry* that contains information about the rule to be inserted. The programmer must fill the fields of this structure with values required to define his or her rule before passing the pointer as parameter to the function. *rulenum* is an integer value defined the position in the chain of rules where the old rule will be replaced by the new one. Rule numbers start at 1 for the first rule. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.4. iptc_append_entry

Name: iptc_append_entry

Usage: Append a new rule in a chain.

Prototype: int iptc_append_entry(const ipt_chainlabel chain, const struct ipt_entry *e, iptc_handle_t *handle)

Description: This function append a rule defined in structure type *ipt_entry* in chain *chain* (equivalent to insert with rulenum = length of chain).

Parameters: *chain* is a char pointer to the name of the chain to be modified; *e* is a pointer to a structure of type *ipt_entry* that contains information about the rule to be appended. The programmer must fill the fields of this structure with values required to define his or her rule before passing the pointer as parameter to the function. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.5. iptc_delete_num_entry

Name: iptc_delete_num_entry

Usage: Delete a rule in a chain.

Prototype: int iptc_delete_num_entry(const ipt_chainlabel chain, unsigned int rulenum, iptc_handle_t *handle)

Description: This function delete the entry rule in chain *chain* positioned at *rulenum*. Rule numbers start at 1 for the first rule.

Parameters: *chain* is a char pointer to the name of the chain to be modified; *rulenum* is an integer value defined the position in the chain of rules where the rule will be deleted. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.6. iptc_flush_entries

Name: iptc_flush_entries

Usage: Empty a chain.

Prototype: int iptc_flush_entries(const ipt_chainlabel chain, iptc_handle_t *handle)

Description: This function flushes the rule entries in the given chain (ie. empties chain).

Parameters: *chain* is a char pointer to the name of the chain to be flushed; *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.7. iptc_zero_entries

Name: iptc_zero_entries

Usage: Zeroes the chain counters.

Prototype: int iptc_zero_entries(const ipt_chainlabel chain, iptc_handle_t *handle)

Description: This function zeroes the counters in the given chain.

Parameters: *chain* is a char pointer to the name of the chain which counters will be zero; *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.8. iptc_create_chain

Name: iptc_create_chain

Usage: Create a new chain.

Prototype: int iptc_create_chain(const ipt_chainlabel chain, iptc_handle_t *handle)

Description: This function create a new chain in the table.

Parameters: *chain* is a char pointer to the name of the chain to be created; *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.9. iptc_delete_chain

Name: iptc_delete_chain

Usage: Delete a chain.

Prototype: int iptc_delete_chain(const ipt_chainlabel chain, iptc_handle_t *handle)

Description: This function delete the chain identified by the char pointer *chain* in the table.

Parameters: *chain* is a char pointer to the name of the chain to be deleted; *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.10. iptc_rename_chain

Name: iptc_rename_chain

Usage: Rename a chain.

Prototype: int iptc_rename_chain(const ipt_chainlabel oldname, const ipt_chainlabel newname, iptc_handle_t *handle)

Description: This function rename the chain identified by the char pointer *oldname* to a new name *newname* in the table.

Parameters: *oldname* is a char pointer to the name of the chain to be renamed, *newname* is the new name; *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.11. iptc_set_policy

Name: iptc_set_policy

Usage: Set the policy in a built–in chain.

Prototype: int iptc_set_policy(const ipt_chainlabel chain, const ipt_chainlabel policy, struct ipt_counters *counters, iptc_handle_t *handle)

Description: This function set the policy in chain *chain* to the value represented by the char pointer *policy*. If you want to set at the same time the counters of the chain, fill those values in a structure of type *ipt_counters* and pass a pointer to it as parameter *counters*. Be careful: the chain *must be* a built–in chain.

Parameters: *chain* is a char pointer to the name of the chain to be modified; *policy* is a char pointer to the name of the policy to be set. *counters* is a pointer to an *ipt_counters* structure to be used to set the counters of the chain. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.12. iptc_zero_counter

Name: iptc_zero_counter

Usage: Zero counters of a rule in a chain.

Prototype: int iptc_zero_counter(const ipt_chainlabel chain, unsigned int rulenum, iptc_handle_t *handle)

Description: This function zero packet and byte counters of the entry rule in chain *chain* positioned at *rulenum*. Rule numbers start at 1 for the first rule.

Parameters: *chain* is a char pointer to the name of the chain to be modified; *rulenum* is an integer value defined the position in the chain of rules of the rule which counters will be zero. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

12.13. iptc_set_counter

Name: iptc_set_counter

Usage: Set counters of a rule in a chain.

Prototype: int iptc_set_counter(const ipt_chainlabel chain, unsigned int rulenum, struct ipt_counters *counters, iptc_handle_t *handle)

Description: This function set packet and byte counters of the entry rule in chain *chain* positioned at *rulenum* with values passed in a type structure *ipt_counters*. Rule numbers start at 1 for the first rule.

Parameters: *chain* is a char pointer to the name of the chain to be modified; *rulenum* is an integer value defined the position in the chain of rules of the rule which counters will be set. *counters* is a pointer to an *ipt_counters* structure to be used to set the counters of the rule; the programmer must fill the fields of this structure with values to be set. *handle* is a pointer to a structure of type *iptc_handle_t* that was obtained by a previous call to *iptc_init*.

Returns: Returns integer value 1 (true) if successful; returns integer value 0 (false) if fails. In this case *errno* is set to the error number generated. Use *iptc_strerror* to get a meaningful information about the problem. If errno == 0, it means there was a version error (ie. upgrade *libiptc*).

13. Bandwidth meter

In this chapter I am going to develop a simple bandwidth meter using the following functions from *libiptc*:

- To initialize the system: *iptc_handle_t iptc_init(const char *tablename)*.
- To catch from errors: *const char *iptc_strerror(int err)*.
- To iterate through the chains of the table: *const char *iptc_first_chain(iptc_handle_t *handle)* and *const char *iptc_next_chain(iptc_handle_t *handle)*.
- To read packet and byte counters for a specific rule: *struct ipt_counters *iptc_read_counter(const ipt_chainlabel chain, unsigned int rulenum, iptc_handle_t *handle)*.

Also the function *gettimeofday* will be used to measure elapsed time and the function *getopt* to get options from the command line.

I don't know really if the term *bandwidth meter* is well used here. I interpret *bandwidth* as a reference to a flow capacity; perhaps a better term could be *flow meter*.

Here is the *bandwidth meter* `bw.c`. It's well commented to be easy followed by everyone:

```
/*
 * How to use libiptc- program #4
 * /usr/local/src/bw.c
 * By Leonardo Balliache - 04.09.2002
 * e-mail: leonardo@opalsoft.net
 * --WELL COMMENTED-- to be easy followed by everyone.
 */

/* include files required */
#include <getopt.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h>
#include <time.h>
#include <unistd.h>
#include "libiptc/libiptc.h"
#include "iptables.h"

/* colors to differentiate chains measures */
#define RED     "\033[41m\033[37m"
#define GREEN   "\033[42m\033[30m"
#define ORANGE  "\033[43m\033[30m"
#define BLUE    "\033[44m\033[37m"
#define MAGENTA "\033[45m\033[37m"
#define CYAN    "\033[46m\033[30m"
#define WHITE   "\033[47m\033[30m"
#define BLACK   "\033[40m\033[37m"
#define RESET   "\033[00m"

/* maximum number of chains to be processed */
#define MAXUSERCHAINS 7

/* time between measures in seconds; adjust as you like */
```

```
#define SLEEPTIME 1

/* structure to count bytes per chain */
struct bwcnt  {
  int start;           /* the chain was initialized */
  u_int64_t icnt;      /* bytes through; previous measure */
  u_int64_t ocnt;      /* bytes through; current measure */
  double bw;           /* bandwitdh (flow) on this chain */
};

/* function to calculate differences of time in seconds.
 * micro-seconds precision.
 */
double delta(struct timeval a, struct timeval b)
{
  if (a.tv_usec & b.tv_usec)  {
    a.tv_sec--;
    a.tv_usec += 1000000;
  }
  return a.tv_sec-b.tv_sec + (a.tv_usec-b.tv_usec)/1000000.0;
}

/* main function */
int main(int argc, char *argv[])
{
  int i, j, ok;
  double totbw;
  iptc_handle_t h;
  int c, act_bw = 0;
  const char *chain = NULL;
  const char *tablename = "filter";
  struct timeval ti, to;
  struct bwcnt bw[MAXUSERCHAINS];
  struct ipt_counters *counters;
  char *col[9] = { RED,GREEN,ORANGE,BLUE,MAGENTA,CYAN,WHITE,BLACK,RESET };

  program_name = "bw";
  program_version = NETFILTER_VERSION;

 /* check options
  * we have 2 options:
  *        -c = display current flow (each SLEEPTIME).
  *        -a = display average flow (from start); default option.
  */
  while ((c = getopt (argc, argv, "ac")) != -1)
  switch (c)  {
  case 'a':
    act_bw = 0;
    break;
  case 'c':
    act_bw = 1;
    break;
  case '?':
    if (isprint(optopt))
      fprintf (stderr, "Unknown option `-%c'.\n", optopt);
    else
      fprintf (stderr,"Unknown option character `\\x%x'.\n",optopt);
    exit(1);
  default:
    abort();
  }
```

```
/* initialize array of chains */
memset(&bw, 0, MAXUSERCHAINS * sizeof(struct bwcnt));

/* get time to start meter on variable ti */
gettimeofday(&ti, NULL);

/* fire meter loop */
if ( act_bw )
  printf("Displaying current flow values ...\n");
else
  printf("Displaying average flow values ...\n");

/* forever loop; abort the program with ^C */
while ( 1 )  {
  /* you have to initialize for each measure to be done */
  if ( !(h = iptc_init(tablename)) )  {
    printf("Error initializing: %s\n", iptc_strerror(errno));
    exit(errno);
  }
  ok = 0;    /* we start a new loop */
  gettimeofday(&to, NULL);  /* have a time shoot */

  /* iterate through each chain of the table */
  for (chain = iptc_first_chain(&h), i = 0;
       chain;
       chain = iptc_next_chain(&h))  {
    if ( iptc_builtin(chain, h) )
      continue;    /* if it is a built-in chain, ignore it */

    /* ok, read the counters of this chain */
    if ( !(counters = iptc_read_counter(chain, 0, &h)) )  {
       printf("Error reading %s: %s\n", chain, iptc_strerror(errno));
       exit(errno);
    }

    /* check that we do not have more chains than we can process */
    if ( i >= MAXUSERCHAINS )  {
       printf("Maximum of %d user chains exceeded!!\n", MAXUSERCHAINS);
       exit(1);
    }

    /* this chain counter has not been initialized; initialize it */
    if ( bw[i].start == 0 )  {
      bw[i].icnt = counters->bcnt;
      bw[i].start = 1;
    }

    /* this chain has a previous measure; take the current one */
    else  {
      bw[i].ocnt = counters->bcnt;
      if ( bw[i].ocnt == bw[i].icnt )    /* no new bytes flowing? */
        bw[i].bw = 0;                    /* flow is zero */
      else
       /* flow in this chain is:
        *   current bytes count (bw[i].octn)    *minus*
        *   previous bytes count (bw[i].icnt)   *divided by*
        *   128.0 to convert bytes to kbits     *and divided by*
        *   difference in times in seconds      *to get*
        *   flow in kbits/sec that is what we want.
        */
        bw[i].bw = (bw[i].ocnt - bw[i].icnt) / (128.0 * delta(to, ti));
```

```
        /* do you want current flow of this chain? initialize previous
         * bytes count to current bytes count; we get the flow in last
         * SLEEPTIME elapsed time.
         */
         if ( act_bw )
           bw[i].icnt = bw[i].ocnt;
         ok = 1;    /* ok, we have some measure to show */
       }
       ++i;  /* next chain, please */
     }

     /* we iterate and i == 0; we do not have user chains at all */
     if ( i == 0 )  {
        printf("No user chains to meter!!\n");
        exit(1);
     }

    /* do you want to measure current flow? initialize previous time
     * to actual time; we get the time elapsed in last SLEEPTIME.
     */
     if ( act_bw )
       ti = to;

     /* do we have something to show? ok, display it */
     if ( ok )  {
       totbw = 0;
       for ( j = 0; j < i; ++j )  {
         totbw = totbw + bw[j].bw;   /* calculate total flow */
       }
       printf("%s%6.1fk:%s ", col[7], totbw, col[8]);  /* display total */
       for ( j = 0; j < i; ++j )  {  /* display flow of each chain in color */
         printf("%s%6.1fk%s ", col[j], bw[j].bw, col[8]);
       }
       printf("\n");
     }
     sleep(SLEEPTIME);  /* rest a little; you go too fast */
   }             /* give us enough time in order to let some bytes flow */

   exit(0);  /* bye, we have our measures!! */

} /* main */
```

Write your program and compile as before:

```
bash# ./ipt-cc bw
```

Before using the meter we need to set our environment.

First, we have to have at least 2 PCs connected in a network. This is our diagram configuration:

```
+--------+ eth0       eth0 +--------+
| PC #1  +-----------------+ PC #2  |
+--------+                 +--------+
eth0=192.168.1.1           eth0=192.168.1.2
```

Second, we need to install a very nice and useful package called *netcat* written by Hobbit. This *excellent* package will help us to inject and receive a flow of bytes between 2 NICs. If you don't have the package in your system, download it from http://rr.sans.org/audit/netcat.php.

The version that I use is *1.10−277*. To install it follow these instructions:

```
bash# cp netcat-1.10.tar.gz /usr/local/src
bash# tar xzvf netcat-1.10.tar.gz
bash# cd netcat-1.10
```

My version requires to make a patch first; check yours if you have a file with a *.dif* extension and apply it too:

```
bash# patch -p0 -i netcat-1.10.dif
```

Next compile the package using *make*:

```
bash# make linux
```

Copy the binary *nc* to your user bin directory:

```
bash# cp nc /usr/bin
```

And also to the second PC in your network:

```
bash# scp nc 192.168.1.2:/usr/bin
```

We are going to use *netcat* to "listen" to a flow of bytes from PC #2 and to "talk" from PC #1. Using tty1 to tty4 consoles on PC #2 let's start *netcat* to listen from this PC. Go to PC #2 and in tty1 type:

```
bash# nc -n -v -l -s 192.168.1.2 -p 1001 >/dev/null
```

netcat must respond with:

```
listening on [192.168.1.2] 1001 ...
```

This command started *netcat* to listen from address *192.168.1.2* using port number *1001*. Arguments are: *−n* = use numeric address identification; *−v* = verbose; *−l* = listen. All the flow that *netcat* receives in *192.168.1.2:1001* will be redirected to the "black hole" in `/dev/null`.

Repeat the command in tty2, tty3 and tty4; change to tty2 using **ALT−F2** and after logging in write:

```
bash# nc -n -v -l -s 192.168.1.2 -p 1002 >/dev/null
```

Now we are "listening" to the same address but port number *1002*.

Go on now with tty3:

```
bash# nc -n -v -l -s 192.168.1.2 -p 1003 >/dev/null
```

And tty4:

```
bash# nc -n -v -l -s 192.168.1.2 -p 1004 >/dev/null
```

Now we are listening in PC #2, address *192.168.1.2* in ports *1001*, *1002*, *1003* and *1004*.

Come back to PC #1 and let's set the environment to allow *iptables* to help us to complete our tests:

On PC #1, type the into tty1 as follows:

```
bash# iptables -F
bash# iptables -X
bash# iptables -N chn_1
bash# iptables -N chn_2
bash# iptables -N chn_3
bash# iptables -N chn_4
bash# iptables -A chn_1 -j ACCEPT
bash# iptables -A chn_2 -j ACCEPT
bash# iptables -A chn_3 -j ACCEPT
bash# iptables -A chn_4 -j ACCEPT
bash# iptables -A OUTPUT -o eth0 -p tcp --dport 1001 -j chn_1
bash# iptables -A OUTPUT -o eth0 -p tcp --dport 1002 -j chn_2
bash# iptables -A OUTPUT -o eth0 -p tcp --dport 1003 -j chn_3
bash# iptables -A OUTPUT -o eth0 -p tcp --dport 1004 -j chn_4
```

These commands will:

- Flush all chains in table *filter*.
- Delete all user chains in table *filter*.
- Create user chains *chn_1*, *chn_2*, *chn_3* and *chn_4*.
- Establish a target *ACCEPT* in each user chain.
- Create 4 rules in chain *OUTPUT* that matches port numbers *1001* to *1004* and target it to user chains *chn_1* to *chn_4*.

Now start the *bw* meter using current values:

```
bash# ./bw -c
```

It must respond with:

```
Displaying current flow values ...
   0.0k:    0.0k    0.0k    0.0k    0.0k
   0.0k:    0.0k    0.0k    0.0k    0.0k
   0.0k:    0.0k    0.0k    0.0k    0.0k
   0.0k:    0.0k    0.0k    0.0k    0.0k
```

It informs that measures are current flows. Every line is a measure taken each *SLEEPTIME* lapse (1 second in our program). First column (in black) are total flow, next columns (in red, green, orange and blue) are flows in chains *chn_1*, *chn_2*, *chn_3* and *chn_4* respectively. Of course we do not have any flow now. However let *bw* run and continue reading.

Let's start now one of our byte flows; go to tty2 in PC #1 with **ALT−F2** and after logging in, type:

```
bash# yes 0000000000000000 | nc -n -v -s 192.168.1.1 -p 2001 192.168.1.2 1001
```

netcat responds with:

```
(UNKNOWN) [192.168.1.2] 1000 (?) open
```

Now we have a flow of bytes from PC #1 to PC #2. *yes* generates a constant flow of zeroes; this flow is piped to *netcat* through address *192.168.1.1*, port *2001* and sends it to PC #2, address *192.168.1.2*, port *1001* (where PC #2 is listening).

Check now the display of *bw* in tty1:

```
7653.2k: 7653.2k    0.0k    0.0k    0.0k
7829.5k: 7829.5k    0.0k    0.0k    0.0k
7786.7k: 7786.7k    0.0k    0.0k    0.0k
7892.1k: 7982.1k    0.0k    0.0k    0.0k
```

Your mileage can vary depending of the physical characteristics of your system. In mine I have a flow of aproximately 7700 kbits/sec in the first chain *chn_1* which corresponds to port number *1001* in PC #2.

Let's start now the second bytes flow; go to tty3 in PC #1 with **ALT−F3** and after logging in, type:

```
bash# yes 0000000000000000 | nc -n -v -s 192.168.1.1 -p 2002 192.168.1.2 1002
```

netcat responds with:

```
(UNKNOWN) [192.168.1.2] 1002 (?) open
```

Now we have 2 flows of bytes from PC #1 to PC #2; one from *192.168.1.1:2001* to *192.168.1.2:1001* and another from *192.168.1.1:2002* to *192.168.1.2:1002*.

Now check the display of *bw* in tty1:

```
7819.6k: 4144.2k 3675.4k    0.0k    0.0k
8090.5k: 3923.9k 4166.6k    0.0k    0.0k
7794.7k: 3920.8k 3873.9k    0.0k    0.0k
7988.3k: 3754.6k 4233.7k    0.0k    0.0k
```

Now we have 2 flows; each of them is more or less 50% of the total flow going out of the computer. The Linux kernel tries to balance the bandwidth available between the 2 channels of output.

To continue, start the 2 aditional flows through channels *192.168.1.1:2003−192.168.1.2:1003* and *192.168.1.1:2004−192.168.1.2:1004*.

In tty4 type:

```
bash# yes 0000000000000000 | nc -n -v -s 192.168.1.1 -p 2003 192.168.1.2 1003
```

In tty5 type:

```
bash# yes 0000000000000000 | nc -n -v -s 192.168.1.1 -p 2004 192.168.1.2 1004
```

The display of *bw* in tty1 will be something like:

```
8120.6k: 1705.3k 2354.9k 1898.6k 2161.8k
7765.3k: 1634.2k 2560.2k 2011.4k 1559.5k
7911.9k: 1699.8k 2090.3k 1768.0k 2353.8k
```

```
8309.4k: 1734.5k 1999.7k 1999.9k 2575.3k
```

Total bandwidth is distributed between the 4 channels of flow.

14. Controlling flows

In this chapter we are going to try to control the flows using the Linux kernel queue disciplines. Perhaps, depending on how you compiled your kernel, you will again need to run **make menuconfig**, re−configure your options, re−compile and re−install your kernel.

This chapter *is not* and *does not pretend to be* a tutorial about the implementation of *QoS (Quality of Service)* in Linux. If you don't have previous experience with *QoS* it's better to read some references at the end of this document to acquire the concepts required for *QoS* implementation.

With this advice, I'm not going to explain in detail each of the commands needed to control flows in Linux because it is not the goal of this HOWTO. However, the implementation of some of these techniques will serve us to show the bandwidth meter (based on *libiptc*) behaviour.

First check if you have QoS implementation options implemented in your kernel. Run **make menuconfig**, follow the menu to *Networking options* and look for last menu of this option *QoS and/or fair queueing*. Here use (or check if they are active) these options:

```
       [*] QoS and/or fair queueing
       <M> CBQ packet scheduler
       <M> CSZ packet scheduler
       [*] ATM pseudo-scheduler
       <M> The simplest PRIO pseudoscheduler
       <M> RED queue
       <M> SFQ queue
       <M> TEQL queue
       <M> TBF queue
       <M> GRED queue
       <M> Diffserv field marker
       <M> Ingress Qdisc
       [*] QoS support
       [*]   Rate estimator
       [*] Packet classifier API
       <M>   TC index classifier
       <M>   Routing table based classifier
       <M>   Firewall based classifier
       <M>   U32 classifier
       <M>   Special RSVP classifier
       <M>   Special RSVP classifier for IPv6
       [*]   Traffic policing (needed for in/egress)
```

Save your configuration, recompile your kernel and modules, and re−install it. We are going to use the *CBQ packet scheduler* to implement some queues to control bytes flow in our PC #1 NIC.

Personally I preferred the excellent *HTB queueing discipline implementation* by Martin Devera but actually this implementation is not in standard Linux (but it will be); for implementing it you have to patch your kernel before recompiling and it's better not to complicate things more. However I have to say that this queue discipline is a lot more simple to use than *CBQ* happens to be. More information on *HTB queueing discipline* are linked at the end of this document.

Having compiled and re−installed your kernel you have to install the *iproute2* package that will be used to run the commands needed to implement the queues. Download this package from ftp://ftp.inr.ac.ru/ip−routing.

I'm working with version *2.2.4−now−ss001007*. To install it follow these instructions:

```
bash# cp iproute2-2.2.4-now-ss001007.tar.gz /usr/local/src
bash# tar xzvf iproute2-2.2.4-now-ss001007.tar.gz
bash# cd iproute2
bash# make
```

After *make* compiles the *iproute2* package successfully the *ip* utility will be in `iproute2/ip` directory and the *tc* utility in `iproute2/tc` directory. Copy both of them to `/usr/bin` directory:

```
bash# cp ip/ip /usr/bin
bash# cp tc/tc /usr/bin
```

Now, using the *tc* utility, we are going to create a *CBQ* queue in the interface *eth0* of the PC #1 computer. This queue will have 4 classes as children and each of these classes will be used to control the 4 flows from *192.168.1.1* to *192.168.1.2* through ports *1001* to *1004*.

Write and run the following commands:

```
bash# tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 10Mbit \
avpkt 1000 cell 8
```

This command creates the main (root) cbq queue 1:0 in the *eth0* interface; the bandwidth of this queue is 10Mbit/sec corresponding to our Ethernet interface.

Now write and run:

```
bash# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 10Mbit \
rate 1000kbit prio 8 allot 1514 cell 8 maxburst 20 avpkt 1000 bounded
```

This command create the main cbq class 1:1. The rate of this class will be 1000kbit/sec.

Now we are going to create 4 classes ownned by this class; the classes will have rates of 100kbit, 200kbit, 300kbit and 400kbit respectively. Write and run these commands:

```
bash# tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 10Mbit \
rate 100kbit prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000

bash# tc class add dev eth0 parent 1:1 classid 1:4 cbq bandwidth 10Mbit \
rate 200kbit prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000

bash# tc class add dev eth0 parent 1:1 classid 1:5 cbq bandwidth 10Mbit \
rate 300kbit prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000

bash# tc class add dev eth0 parent 1:1 classid 1:6 cbq bandwidth 10Mbit \
rate 400kbit prio 5 allot 1514 cell 8 maxburst 20 avpkt 1000
```

Each of these classes will have a *sfq* queue discipline attached to them to dispatch their packets. Write and run these commands:

```
bash# tc qdisc add dev eth0 parent 1:3 handle 30: sfq perturb 15
bash# tc qdisc add dev eth0 parent 1:4 handle 40: sfq perturb 15
bash# tc qdisc add dev eth0 parent 1:5 handle 50: sfq perturb 15
bash# tc qdisc add dev eth0 parent 1:6 handle 60: sfq perturb 15
```

These commands create 4 *sfq* queue disciplines, one for each class. *sfq* queue discipline is some kind of *fair controlling queue*. It tries to give to each connection in an interface same oportunity to their packets to be dispatched to at all.

Finally we are going to create filters to assign flows to ports *1001*, *1002*, *1003* and *1004* to classes *1:3*, *1:4*, *1:5* and *1:6* respectively. Write and run as follows:

```
bash# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
dport 1001 0xffff flowid 1:3

bash# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
dport 1002 0xffff flowid 1:4

bash# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
dport 1003 0xffff flowid 1:5

bash# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
dport 1004 0xffff flowid 1:6
```

After running all these commands, now check your *bw* meter (you must be running *netcat* listening at ports *1001* to *1004* in PC #2 and talking in PC #1 as was explained in previous chapter and *bw* running in *current −c* mode). You will have something like this:

```
Current flow values ...
   1099.9k:  108.8k  196.5k  337.9k  456.8k
   1104.2k:  115.3k  184.9k  339.9k  464.1k
   1102.1k:  117.3k  174.7k  339.7k  470.5k
   1114.4k:  113.6k  191.7k  340.7k  468.4k
   1118.4k:  113.7k  194.3k  340.5k  469.9k
```

bw show us how flows are controlling using queue disciplines of the Linux kernel. As you see, *CBQ queue discipline* is not a very precise queue but you more or less have a flow of approximately *1000=100+200+300+400* on interface *eth0*.

To step back, write and run as follows:

```
bash# tc qdisc del dev eth0 root handle 1:0 cbq
```

on PC #1, to delete the main (root) queue discipline and owned classes and filters.

```
bash# killall nc
```

on PC #2 and PC #1, to stop *netcat*.

```
bash# iptables -F
bash# iptables -X
```

on PC #1, to clear *iptables* rules and chains.

```
bash# Ctrl-C
```

on PC #1, tty1 to stop *bw* bandwidth meter.

15. Some interesting links

1. iptables−1.2.6 by Paul Russell.
2. Linux netfilter Hacking HOWTO by Paul Russell.
3. iproute2 by Alexey Kuznetsov.
4. Advance routing Linux HOWTO.
5. HTB queueing discipline implementation by Martin Devera.
6. Linux−Advance Networking Overview by Saravanan Radhakrishnan.
7. monitor.pl by Stef Coene.
8. netcat by Hobbit.

16. About the author

Leonardo Balliache is a power electrical engineer that left high voltage lines, transformers and protection relays in 1983 to dedicated full of his time to computer sciences.

He is the General Manager of OpalSoft, a venezuelan company dedicated to business packages software development.

In 1989 he started learning Unix using Coherent operating system. After this he was interested in Linux and specially in bandwidth bottleneck problems, bandwidth controlling, packet filtering and hierarching, Linux QoS (Quality of Service), advanced routing, network protection, firewalling, private network connection through the Internet and solving line and server load balancing problems.

His company will be opening a new area of business offering Linux QoS solution implementations in Venezuela.

Married to Cielo, with 3 sons (Jose, Dario, Gustavo), he can be reached at leonardo@opalsoft.net. He is working now (please be patient) to open a QoS Linux information site at http://opalsoft.net/qos/ to interchange knowledge with people interested and to make his works in the Linux "best of all" operating system available to the public.

April 30, 2002

Caracas, Venezuela