

Writing Netfilter modules

Modified Version

JAN ENGELHARDT, NICOLAS BOULIANE

rev. September 15, 2011

The Netfilter/Xtables/iptables framework gives us the possibility to add features. To do so, we write kernel modules that register against this framework. Also, depending on the feature's category, we write an iptables userspace module. By writing your new extension, you can match, mangle, track and give faith to a given packet or complete flows of interrelated connections. In fact, you can do almost everything you want in this world. Netfilter is merely a series of hooks in various points in a protocol stack. Beware that a little error in a kernel module can crash the computer.

We will explain the skeletal structures of Xtables and Netfilter modules with complete code examples and by this way, hope to make the interaction with the framework a little easier to understand. We assume you already know a bit about iptables and that you do have C programming skills.

Copyright © 2005 Nicolas Bouliane <acidfu (at) people.netfilter.org>,
Copyright © 2008–2011 Jan Engelhardt <jengelh (at) medozas.de>,
Revision by Juan Antonio Osorio Robles <jaosorior (at) gmail.com>

This work is made available under the Creative Commons Attribution-Noncommercial-Sharealike 3.0 (CC-BY-NC-SA) license. See <http://creativecommons.org/licenses/by-nc-sa/3.0/> for details. (Alternate arrangements can be made with the copyright holder(s).)

Additionally, modifications to this work must clearly be indicated as such, and the title page needs to carry the words “Modified Version” if any such modifications have been made, unless the release was done by the designated maintainer(s). The Maintainers are members of the Netfilter Core Team, and any person(s) appointed as maintainer(s) by the coreteam.

About the authors

Nicolas has been using Linux since 1998. He is the co-author of the Netfilter geoip module. He is currently working for AirJaldi and his current focus is to empower communities through wireless technologies using free software. He wrote the original “How to write your own iptables module” in February 2005.

Jan is a consultant for systems and network administration with a strong focus on Linux-based environments. He uses Linux since fall 1999, and is involved in kernel development since 2003. His latest significant activities are in the field of firewalling with Netfilter and Xtables. Nicolas’s article has been extensively rewritten and extended by Jan in 2008, up to a book’s size.

Acknowledgments

I (Jan) would like to thank Nicolas for the original version of this document. I like to pass on the knowledge, but had hesitated before to write it down all myself, and so, Nicolas’s earlier version inspired me to update and extend it.

Jan Engelhardt, 2008-January-08

State of the book

Two sections are incomplete, these are sections 5.7 and 9. Improvements are welcome.

This book is designed for, and made available in ISO A4 format, for the simple reason that most regular printed books have anything *but* a standard size.

Contents

I	Xtables	4
1	Nomenclature	4
2	Match extensions	5
3	Userspace plugin	18
4	Tricks and traps	28
5	Target extensions	35
II	Connection Tracking	46
6	nf_conn structure	46
7	Layer-3 connection tracker	48
8	Layer-4 connection tracker	50
9	Connection tracking helper	55
III	Netfilter Core	57
10	Netfilter Hooks	57
IV	Appendix	60
A	Function reference	60
B	Tricks and traps	62
C	Kernel/Xtables-addons differences	62

Part I

Xtables

Xtables is about the table-based firewalling that you most likely know from running the `iptables(8)` command. In this part, the structure of modules usable with `iptables` will be described.

This version of the book focuses on the Netfilter API as found in Linux 2.6.35–2.6.38.

For `iptables`, at least version 1.4.3 is needed, because it contains the necessary initial infrastructure to support external (out-of-tree) `iptables` extensions. This book will only concentrate on recent API version(s) only however, which is `libxtables.so.4` (starting from `iptables` 1.4.6) as of this writing. It is deemed that upgrading `iptables` is much easier than to upgrade the kernel. Since `iptables` 1.4.2, a lot of code has been moved into a shared library, whose “so-version” basically indicates the API versions supported. This API version is decoupled from the `iptables` release version.

Jan released a new package called `Xtables-addons` at the end of January 2008 that replaces the old `patch-o-matic(-ng)` where you can easily add new extensions without having to bother about the build infrastructure complexities or horrid patching strategies. It also provides API compatibility glue code so that you can write extensions using (a slight modification of) the latest kernel API and get it running on older kernels. Currently, `Xtables-addons` has glue code to provide backwards compatibility down to 2.6.17, which is a coverage of 4½ years to date.

This book covers a substantial amount of module code. If you would like to get started with pre-existing code instead of trying to write your module from scratch, get ahold of the `Xtables-addons` package, or a clone of its git repository. The repository also has a branch “demos” which contains the sample module `xt_ipaddr` which is not included in the normal tarball.

See <http://xtables-addons.sf.net/> for details and sources.

1 Nomenclature

`x_tables` refers to the kernel module that provides the generic, (mostly) protocol-independent table-based firewalling used in Linux, and `ip_tables`, `ip6_tables`, `arp_tables` and `ebtables` are the kernel modules providing family-specific tables for the `iptables`, `ip6tables`, `arptables` and `ebtables` tools.

By convention, names of Xtables matches are always lower-case, and names of Xtables targets are upper-case. This is not a hard rule, though. In fact, `Ebtables` used to use lower-case target names, e. g. `mark_m` for the match, `mark` for the target. The choice to go for upper-casing, or using suffixes, is largely a historic decision for (now-)historic reasons, and may limitations have been lifted since then, and in fact, for new modules, it is requested that their filename be lowercase to avoid clashes on case-insensitive filesystems.

Xtables module names are prefixed with `xt_`, forming, for example, `xt_connmark` for the connmark match. `ip`, `ip6`, `arp` and `eb` table modules traditionally used distinct prefixes, according to their subsystem. They were `ipt_`, `ip6t_`, `arpt_` and `ebt_`, respectively. Use of these is discouraged and is to be avoided for new modules. Today, these prefixes only survive as aids for directed kernel module loading in module aliases (similar to “net-pf-2-protocol-132” and “net-pf-10-protocol-132” for `setp.ko`).

```
MODULE_ALIAS("ipt_mymatch");
MODULE_ALIAS("ip6t_mymatch");
MODULE_ALIAS("arpt_mymatch");
MODULE_ALIAS("ebt_mymatch");
```

Add lines like these to the source code of your module — ideally only those which you support. If your module does not match on ARP packets for example, do not add an `arpt_` alias. Of course, use upper-case as mentioned earlier for targets.

Filenames are usually set forth by their logical module name, i.e. `xt_mymatch.c`. Originally, each match and each target was put into its own source file, but since the module overhead is not negligible after all, a number of extensions are bundled into one module. Examples from Linux 2.6.37 are `xt_mark.c`, `xt_connmark.c`, `xt_HL.c`, `xt_DSCP.c`. Reasons for this are their close nature, reducing compile time, reducing memory footprint on modular kernels, and the avoidance of filenames that only differ in case, which are known to be a trouble on case-insensitive filesystems. As long as it has the appropriate `MODULE_ALIASES` to ensure it can be loaded — if it is not already statically compiled into the kernel — there is no hard requirement on the filename.

As far as userspace is concerned, iptables modules use `libxt_` as prefix, and modules must adhere to it because both the Makefiles and the iptables codebase responsible for loading plugins have it hardcoded on it.

2 Match extensions

The duty of a match module is to inspect each packet received and to decide whether it matches or not, according to our criteria. The criteria can be quite anything you can think of, though there are limits. The most obvious matches are of course match by source or destination address, which is done inside Xtables rather than a separate match, and source and/or destination port for TCP/UDP (and/or others), which on the other hand, is actually done inside a match. (This is because TCP is layer 4 already, while the IP address is with layer 3.) There are also advanced modules, such as `xt_connlimit` to match on concurrent number of connections. Combined with exotics such as `xt_time` (to match on the system time), daytime-based connection limits could be enforced, for example. Many matches are simple piece of code, others require a bit more code for their housekeeping (such as `xt_hashlimit`).

A match generally may not modify much — as you will later see from the function prototypes, the `skb` is marked `const`, as are other variables. Modifying any of this data should only be done in targets, which are discussed later, but again, there are exceptions. In fact, the boundary between match and target is fading a bit.

In this chapter, we will be writing a simple IP address match (even if Xtables does a better job at it) called “`xt_ipaddr`”.

2.1 Header file

The header file for our match is where you describe the binary interface between userspace and the kernel module. We usually begin with the header file because it is the first thing that comes to mind — when you ask yourself “what do I actually want to match (against)?”, and what information needs to be conveyed to the kernel.

As far as our `xt_ipaddr` sample module is concerned, we want to match on source and/or destination address, so we need storage for these, and also a flags field with which we indicate whether to consider source/destination address in the match, or not, and whether (or not) to invert the result. So we have:

```
#ifndef _LINUX_NETFILTER_XT_IPADDR_H
#define _LINUX_NETFILTER_XT_IPADDR_H 1

enum {
```

```

    XT_IPADDR_SRC      = 1 << 0,
    XT_IPADDR_DST      = 1 << 1,
    XT_IPADDR_SRC_INV  = 1 << 2,
    XT_IPADDR_DST_INV  = 1 << 3,
};

```

These are the constant names for the `flags` field. We use `1 << n` here because that is a bit less error prone when initially typing the raw values like `0x04`, `0x08`, `0x10`, like accidentally writing a number which has more than one bit set.

Alternatively, we could have put the invert flags into a separate variable. Such is useful when the match flags are the same as the invert flags. The main code would then use `(invert_flags & XT_IPADDR_SRC)` instead of `(flags & XT_IPADDR_SRC_INV)` to test for inversion, for example. We do not do this in our example however, since flags and invert flags fit into the 8-bit `flags` member, and a split would otherwise take up 16.

You should not use types which do not have a fixed width for the parameter exchange — `short`, `int` and `long` are all taboo! This is because `long` for example has a different size in 32- and 64-bit environments. On x86, `long` is 4 bytes, but on x86_64, it is 8 bytes. If you run a 32-bit `iptables` binary with a 64-bit kernel — and this is very common in the sparc64 world —, problems can arise because the size of the types is not the same on both ends. Instead, use the types listed in table 1.

	char	short	int/long	long/long long
Unsigned	<code>__u8</code>	<code>__u16</code>	<code>__u32</code>	<code>aligned_u64</code>
Signed	<code>__s8</code>	<code>__s16</code>	<code>__s32</code>	<code>__s64</code> <code>__attribute__((aligned(8)))</code>
Unsigned little-endian	<code>__u8</code>	<code>__le16</code>	<code>__le32</code>	<code>aligned_le64</code>
Unsigned big-endian	<code>__u8</code>	<code>__be16</code>	<code>__be32</code>	<code>aligned_be64</code>

Table 1: Fixed types

Note to self: Upcoming change will use `__aligned_u64` (with two leading underscores), `__aligned_be64` and `__aligned_le64`, but has not been merged yet as of v2.6.36-rc3-1437-ga00eac0.

`char` is defined to be of size 1, so it is safe to use. For clarity, use `char` for characters and strings, and `__u8` (or `__s8`) for numbers. 64-bit quantities must be specially aligned, for consistency please do not use `__le64` or `__be64`; for details see section 4.4. There is indeed no `aligned_s64`.

Note that `__le*` and `__be*` are only annotations used in conjunction with `sparse` to flag up possible coding errors. For the C compiler they are equivalent to `__u*`, and you still need to do byteswapping using the appropriate functions (see appendix A).

Try to arrange the members in the struct so that it does not leave any padding holes; this will benefit memory economy.

```

struct xt_ipaddr_mtinfo {
    union nf_inet_addr src, dst;
    __u8 flags;
};

#endif /* _LINUX_NETFILTER_XT_IPADDR_H */

```

`union nf_inet_addr` is a compound introduced in Linux 2.6.25 that can store either an IPv4 or IPv6 address (in various types actually; `ip` is layout-compatible to `in` and `ip6` is compatible to

in6). It is defined in `<linux/netfilter.h>`, and for `struct in_addr` and `struct in6_addr` to work, you need to include `<linux/ip.h>` and `<linux/ipv6.h>` in kernel-space, or `<netinet/in.h>` in userspace, respectively. Xtables-addons provides necessary glue code to make it work on older kernels too.

```
union nf_inet_addr {
    __be32 ip;
    __be32 ip6[4];
    struct in_addr in;
    struct in6_addr in6;
};
```

What address type it actually stored (i.e. how the union should be interpreted) is passed on elsewhere in the Xtables framework. For example, `ip6tables` will always fill `src` and `dst` with IPv6 addresses and only ever calls the `NFPROTO_IPV6` version of the `xt_ipaddr` match. This is also why we will have two separate match functions, one for IPv4 (`ipaddr_mt4`) and one for IPv6 (`ipaddr_mt6`). Of course you could also record the address family inside the struct and instead combine our two match functions. But you would not gain anything from it — usually you cannot combine any code because the IP header is of different type (`struct iphdr` and `struct ipv6hdr`), requiring just as much C code.

2.2 Structural definition

At first, let us look at some basic structures. The `xt_match` structure is defined in `<linux/netfilter/x_tables.h>`. Fields that are not of interest (mostly internal fields like the linked list fields) have been left out here.

```
struct xt_action_param {
    const struct xt_match *match;
    const void *matchinfo;
    const struct net_device *in, *out;
    int fragoff;
    unsigned int thoff;
    unsigned int hook;
    uint8_t family;
    bool hotdrop;
};

struct xt_mtchk_param {
    const char *table;
    const void *entryinfo;
    const struct xt_match *match;
    void *matchinfo;
    unsigned int hook_mask;
    uint8_t family;
};

struct xt_mtdtor_param {
    const struct xt_match *match;
    void *matchinfo;
    uint8_t family;
};
```

```

};

struct xt_match {
    const char name[XT_EXTENSION_MAXNAMELEN];
    uint8_t revision;
    unsigned short family;
    const char *table;
    unsigned int hooks;
    unsigned short proto;

    bool (*match)(const struct sk_buff *skb,
                  struct xt_action_param *);
    int (*checkentry)(const struct xt_mtchk_param *);
    void (*destroy)(const struct xt_mtdtor_param *);

    struct module *me;
};

```

The number of arguments to the functions has grown over time, as have the numbers of extensions, and it became a lengthy job to update all of them whenever an API change was required. Moreover, many extensions do not even use all parameters. Linux 2.6.28(-rc1) thus introduced the parameter structures named `struct xt_*_param` that collect all of the arguments. The `skb` remains outside the structure for the compiler's convenience of applying its optimizations. Xtables-addons uses the same function signatures here.

The types for earlier Linux kernels are numerous of course, but let's just look at the last types that were used directly before 2.6.28:

```

bool (*match)(const struct sk_buff *skb, const struct net_device *in,
              const struct net_device *out, const struct xt_match *match,
              const void *matchinfo, int offset, unsigned int protoff,
              bool *hotdrop);
bool (*checkentry)(const char *table, const void *entry,
                  const struct xt_match *match, void *matchinfo,
                  unsigned int hook_mask);
void (*destroy)(const struct xt_match *match, void *matchinfo);

```

Much has changed since then; `checkentry` also returns `int` as of 2.6.35, just to name an important change.

2.3 Module initialization

We initialize the common fields in the `xt_match` structure. It must not be marked `const`, because it will be added to the chain of a linked list and hence needs to be modifiable. But we will mark it `__read_mostly`, which is yet another of those magic annotation tags that will trigger the linker to specially layout symbols, which actually helps optimizing cachelining(**author?**) [ReadMostly].

```

static struct xt_match ipaddr_mt4_reg __read_mostly = {

```

`name` is the name of the match that you define. `XT_EXTENSION_MAXNAMELEN` is currently 29, so subtracting one for the trailing `'\0'` leaves 28 chars for the name of your match, which

should be enough. `revision` is an integer that can be used to denote a “version” or feature set of a given match. For example, the `xt_multiport` match, which is used to efficiently match up to 15 TCP or UDP ports at a time, supported only 15 source or 15 destination ports in revision 0. Supporting 15 source *and* 15 destination required a change of the private structure, so revision 1 had to be introduced.

Then comes `family`, which specifies the type of family this `xt_match` structure handles. A “family” does not map to a specific layer in the OSI protocol stack as can be seen, but rather for a type of processing. `ip_tables` will only search the Xtables core for extensions with `NFPROTO_IPV4` or `NFPROTO_UNSPEC`; `ip6_tables` only `NFPROTO_IPV6` and `NFPROTO_UNSPEC`; etc. when inserting a new rule. Possible values are shown in table 2.

Constant	Considered by	Value for kernels < 2.6.28
<code>NFPROTO_UNSPEC</code>	(all)	none
<code>NFPROTO_IPV4</code>	<code>ip_tables</code>	<code>PF_INET</code>
<code>NFPROTO_IPV6</code>	<code>ip6_tables</code>	<code>PF_INET6</code>
<code>NFPROTO_ARP</code>	<code>arp_tables</code>	<code>NF_ARP</code>
<code>NFPROTO_BRIDGE</code>	<code>etables</code>	none

Table 2: Possible values for the “`family`” field

Both userspace and kernelspace must agree on the same `<name, revision, address family, size>` 4-tuple for an `xt_match` to be successfully used.

```
.name      = "ipaddr",
.revision  = 0,
.family    = NFPROTO_IPV4,
```

The `table`, `hooks` and `proto` fields can limit where the match may be used. If the field is not provided, no table, hook or protocol restriction will be applied, respectively. There are no Xtables matches we know of that are limited to a specific table, but the field is there for completeness. `hooks` is seen sometimes, for example in `xt_owner`, which matches on the socket sending the skb — information which is only available in the output path as of 2.6.25, so `xt_owner` sets `hooks`. `hooks` will be covered in deeper detail in section 5.3 about Xtables targets.

`proto`, as far as matches are concerned, is primarily used by IPv6 extension header and non-extension layer-4 protocol matches, i.e. when you invoke `iptables -A INPUT -p tcp`, “`-m tcp`” is implicitly added. Since `xt_tcpudp` has `.proto = IPPROTO_TCP`, the module, and hence the entire rule, will only match on TCP traffic. You should not artificially limit a match to a certain protocol, either by use of `proto` or by not interpreting anything else than a specific protocol — please provide code for all protocols, if applicable and possible.

`table` and `proto` are single-value fields, only `hooks` is a bitmask. If you plan to allow a match or target in more than one table — but still not all tables that could possibly exist — or more than one protocol, you need to write an appropriate check in the `checkentry` function (see section 2.6).

The next fields are callbacks that the framework will use. `match` is what is called when a packet is passed to our module, `checkentry` and `destroy` are called on rule insertion and removal, respectively. Since we do not have anything meaningful to do, we will just do a `printk/pr_info` inside `ipaddr_mt_check` and `ipaddr_mt_destroy` in our sample module.

```

        .match      = ipaddr_mt,
        .checkentry = ipaddr_mt_check,
        .destroy    = ipaddr_mt_destroy,

```

Not of less importance is the `matchsize` field which specifies the size of the private structure. Note the details about alignment in section 4.4.

```

        .matchsize = sizeof(struct xt_ipaddr_mtinfo),
        .me        = THIS_MODULE,
};

```

The last line containing `THIS_MODULE` is used for the Linux kernel module infrastructure; among other things, it serves for reference counting, so that the module is not unloaded while a rule exists that references the module. Include `<linux/module.h>` for it.

Your kernel module's init function needs to call `xt_register_match` with a pointer to the struct. This function is called on module loading.

```

static int __init ipaddr_mt_init(void)
{
    return xt_register_match(&ipaddr_mt4_reg);
}

```

When unloading the module, the match needs to be unregistered again.

```

static void __exit ipaddr_mt_exit(void)
{
    xt_unregister_match(&ipaddr_mt4_reg);
}

```

`__init` and `__exit` are markers that cause the functions to be emitted into specific sections in the resulting module (read: linker magic). It does not automatically mean that these are the entry and exit functions. For that, we need the following two extra lines coming after the functions:

```

module_init(ipaddr_mt_init);
module_exit(ipaddr_mt_exit);

```

You should not forget to add the standard module boilerplate, that is, author (you can have multiple lines of them), description and license:

```

MODULE_AUTHOR("Me <my@address.com>");
MODULE_DESCRIPTION("Xtables: Match source/destination address");
MODULE_LICENSE("GPL");

```

Reminding you, just in case you have already forgotten, make sure that the module has the necessary module aliases required for automatic loading.

```

MODULE_ALIAS("ipt_ipaddr");

```

2.4 Naming convention

It is advised to keep symbols (function and variable names) unique across the whole kernel. This is because if you just name your match function “`match`”, which was historically done in a lot of modules (probably due to copy & paste when new modules were developed), it becomes hard to recognize whether it was module1’s match or module2’s match function in a potential kernel stack trace during oops. You do not need to actually use totally-unique names for all symbols, but for the parts that interface to Xtables, it is recommended. The standard naming is the match name, an underscore, “`mt`” (`ipaddr_mt` in our example) and another word for the symbol. You will get to see it in action in the example code as you read through this document. Typically we use these:

- `ipaddr_mt_reg` – the structure (sort of an “object”) containing all the metadata such as name and the function pointer table (“vtable”)
- `ipaddr_mt` – the match function
- `ipaddr_mt_check` – function to check for validity of parameters in our struct
- `ipaddr_mt_destroy` – function to call when rule is deleted
- `struct xt_ipaddr_mtinfo` – structure for our own data
- `ipaddr_mt4` and `ipaddr_mt6`, `ipaddr_mt4_check` and `ipaddr_mt6_check`, etc. – when the IPv4 and IPv6 bits do not share the same entrypoint.
- `struct xt_ipaddr_mtinfoN` – structure for revision *N*

2.5 Point of decision – match function

The Linux networking stack is sprinkled with Netfilter hooks. Thus, when a packet is going to be processed, the networking stack passes the packet to each hook. Of interest here is only Xtables of course; the hooks `NF_IP6_PRI_MANGLE`, `NF_IP6_PRI_FILTER`, and its IPv4 counterparts map to a table. When control is passed to the `ip6t_do_table` function, it will iterate over each rule, which in turn iterates through each match that is used in a given rule. When it is the time for your module to have the packet, it can finally do its job.

```
static bool ipaddr_mt(const struct sk_buff *skb,
                     struct xt_action_param *par)
{
```

The contents of the `xt_action_param` structure have been previously shown in section 2.2, now here is what they are used for. `par->in` and `par->out` are the network devices through which the packet came in or went out; they may be `NULL` in certain chains, see table 3. Interfaces are rarely used in Xtables, only the logging targets seem to make use of it right now.

	in	out
BROUTING, PREROUTING and INPUT	✓	—
FORWARD	✓	✓
OUTPUT and POSTROUTING	—	✓

Table 3: Availability of interface variables

`par->match` points to the structure through which the match function was invoked, and allows to determine which address family was used. If the function was called with the IPv6 structure, which naturally has `.family = NFPROTO_IPV6`, we know that `xt_ipaddr_mtinfo` was filled in with an IPv6 address. This can be helpful when you want to combine code (as is done in `xt_connlimit` for example).

Recent development saw the addition of `par->family` (also available in Linux 2.6.28) which carries the `NFPROTO_*` number the match function was invoked with, so that the family is still known even when `par->match->family == NFPROTO_UNSPEC`.

`par->matchinfo` is the data block copied from userspace, and here we map it. Note that *no* casts are required between `void *` and any other (non-function) pointer type in C, so do not attempt anything unwise!

```
const struct xt_ipaddr_mtinfo *info = par->matchinfo;
```

`skb` contains the packet we want to look at. You need to include `<linux/skbuff.h>` if you do any `skb` operation or access the struct's members. For more information about this powerful structure used everywhere in the Linux networking stack, I recommend the book "Understanding Linux Network Internals" (author?) [LinuxNetInt]. While the latest edition of the book is a few years back and maps to Linux 2.6.14, it is still a very helpful read. There is also section 4.1 in this book.

The `ip_hdr` function returns a pointer to the start of the IPv4 header.

```
const struct iphdr *iph = ip_hdr(skb);
```

Here, we are just printing some of the variables passed to see what they look like. The macros `NIPQUAD_FMT` and `NIPQUAD` are used to display an IPv4 address in readable format. It is defined in `<linux/kernel.h>`.

```
pr_info(
    "xt_ipaddr: IN=%s OUT=%s "
    "SRC=" NIPQUAD_FMT " DST=" NIPQUAD_FMT " "
    "IPSRC=" NIPQUAD_FMT " IPDST=" NIPQUAD_FMT "\n",
    (par->in != NULL) ? par->in->name : "",
    (par->out != NULL) ? par->out->name : "",
    NIPQUAD(iph->saddr), NIPQUAD(iph->daddr),
    NIPQUAD(info->src), NIPQUAD(info->dst));
```

For IPv6 addresses, use `NIP6_FMT` and `NIP6`, respectively. In kernels starting from 2.6.29, printing addresses changed when the `NIP6` macros were removed in favor of the new format characters. When writing a module whilst making use of the Xtables-addons compat layer, continue to use `NIPQUAD/NIP6` and the `_FMT` macros.

```
pr_info("SRC=%pI4 / %pI6\n", &ip4h->saddr, &ip6h->saddr);
```

If the `XT_IPADDR_SRC` flag has been set, we check whether the source address matches the one specified in the rule. If it does not match, the whole rule will not match, so we can already return `false` here. Note that the comparison of `iph->saddr` with `info->src.ip` is XORed with the presence (double exclamation mark) of the inversion flag `XT_IPADDR_SRC_INV` to flip the result of the comparison to get the invert semantics.

```

    if (info->flags & XT_IPADDR_SRC)
        if ((iph->saddr != info->src.ip) ^
            !(info->flags & XT_IPADDR_SRC_INV)) {
            pr_notice("src IP - no match\n");
            return false;
        }

```

For an explanation of the use of “!!”, see appendix B.1.

Here, we do the same, except that we look for the destination address if `XT_IPADDR_DST` has been set.

```

    if (info->flags & XT_IPADDR_DST)
        if ((iph->daddr != info->dst.ip) ^
            !(info->flags & XT_IPADDR_DST_INV)) {
            pr_notice("dst IP - no match\n");
            return false;
        }

```

At the end of the function, we will return `true`, because we have excluded all non-matching cases before **(author?)** [ElseHarmful].

```

    return true;
}

```

If there is a problem that prohibits or makes it impossible to determine whether the packet matched or not, e.g. memory allocation failure or a bogus packet, `par->hotdrop` should be set to `true` and the function should return `false`. Example from `xt_tcpudp`:

```

/* Extract TCP options */
op = skb_header_pointer(skb, par->thoff + sizeof(struct tcphdr),
                        optlen, _opt);

if (op == NULL) {
    par->hotdrop = true;
    return false;
}

```

The `par->thoff` argument to our match function contains the offset into the packet where the transport header for the protocol given in `ipaddr_mt_reg.proto`, in this case the TCP header, begins. `skb_header_pointer` extracts data from the position given in its second argument, which is relative to the `skb->data` pointer. For Ebtables modules, `skb->data` will point to the layer-2 header, whereas for ip, ip6 and arptables, it will be the layer-3 header.

2.6 Rule validation – checkentry function

`checkentry` is often used as a kernel-side sanity check of the data the user input, as you should not rely on the `iptables` userspace program passing in proper information. It is also used to trigger loading of any additional modules that might be required for the match to function properly, such as layer-3 connection tracking, which is essential for connection-based matches like `xt_connlimit`, `xt_conntrack`, and a few more. What’s more, this function may be used to allocate extra memory that may be needed to store state — more on this in section 4.5. This function is called when you try to add a rule, but it happens before the rule is actually inserted.

If you do not plan on loading or verifying anything, you can omit the function.

```

static int ipaddr_mt_check(const struct xt_mtchk_param *par)
{
    const struct xt_ipaddr_mtinfo *info = par->matchinfo;

    pr_info("Added a rule with -m ipaddr in the %s table; this rule is "
           "reachable through hooks 0x%x\n",
           par->table, par->hook_mask);

    if (!(info->flags & (XT_IPADDR_SRC | XT_IPADDR_DST))) {
        pr_info("not testing for anything\n");
        return -EINVAL;
    }

    if (ntohl(info->src.ip) == 0xDEADBEEF) {
        /* This just for fun */
        pr_info("I'm sorry, Dave. I'm afraid I can't let you do that.\n");
        return -EPERM;
    }

    return 0;
}

```

The `checkentry` function may also be used to limit the match to specific tables, hooks or combinations thereof if the mechanisms provided by `struct xt_match` are not sufficient. More on that in section 5.10.

`checkentry` is supposed to return an error code as shown. Often, `-EINVAL` is the most meaningful, but since `EINVAL` is kind of overused in the kernel for whenever there is an invalid option combination or similar, a helpful message should be added.

2.7 Rule destruction – destroy function

The `destroy` function is provided as a counterpart for modules which used `checkentry` as means to load additional modules or allocating space. Of course, we would like to free that space when a rule is removed, and drop additional modules reference count so they can be unloaded if desired. Since our `xt_ipaddr` does not allocate anything or use extra modules, it will just print out something for demonstration. This function may also be omitted.

```

static void ipaddr_mt_destroy(const struct xt_mtdtor_param *par)
{
    const struct xt_ipaddr_mtinfo *info = par->matchinfo;
    pr_info("Test for address %08lX removed\n", info->src.ip);
}

```

2.8 IPv6 support

IPv6 is emerging, and it would be cool if our module also supported IPv6. After all, it is what is supposed to replace IPv4 in the future. Next Header parsing requires a bit more code for IPv6, but since we are just comparing source and destination address in the IPv6 header, our example currently remains small.

If your module inherently does not support IPv6 because, for example, it matches on an IPv4-specific property, you of course do not add match code or a `struct xt_match` for IPv6.

```

static bool ipaddr_mt6(const struct sk_buff *skb,
                      const struct xt_action_param *par)
{
    const struct xt_ipaddr_mtinfo *info = par->matchinfo;
    const struct ipv6hdr *iph = ipv6_hdr(skb);

    if (info->flags & XT_IPADDR_SRC)
        if ((ipv6_addr_cmp(&iph->saddr, &info->src.in6) != 0) ^
            !(info->flags & XT_IPADDR_SRC_INV))
            return false;

    if (info->flags & XT_IPADDR_DST)
        if ((ipv6_addr_cmp(&iph->daddr, &info->dst.in6) != 0) ^
            !(info->flags & XT_IPADDR_DST_INV))
            return false;

    return true;
}

```

You would then declare a new struct `xt_match` with `match` pointing to `ipaddr_mt6`.

```

static struct xt_match ipaddr_mt6_reg __read_mostly = {
    .name      = "ipaddr",
    .revision  = 0,
    .family    = NFPROTO_IPV6,
    .match     = ipaddr_mt6,
    .matchsize = sizeof(struct xt_ipaddr_mtinfo),
    .me        = THIS_MODULE,
};

```

and call `xt_register_match(&ipaddr_mt6_reg)` next to the already existing registration call for `ipaddr_mt4_reg`, of course, with proper error handling:

```

static int __init ipaddr_mt_reg(void)
{
    int ret;

    ret = xt_register_match(&ipaddr_mt4_reg);
    if (ret < 0)
        return ret;
    ret = xt_register_match(&ipaddr_mt6_reg);
    if (ret < 0) {
        xt_unregister_match(&ipaddr_mt4_reg);
        return ret;
    }
    return 0;
}

```

As the number of match structures grow — and the possibility to do revisions just increases the likelihood of that happening — this will accumulate to a great amount of redundantly typed error code paths. There exists a much better way for registering multiple matches at once, which is explained in section 4.2.

2.9 Building the module

To actually build our precious work, we need a Makefile and other bits to make ‘make’ do the right thing. There are a number of different approaches here, in some of which you can skip the build logic largely and concentrate on the module.

2.9.1 Using the Xtables-addons package

Place the modules’ files — `xt_ipaddr.c` and `xt_ipaddr.h` — into the `extensions/` directory and modify the `Kbuild` file to include `xt_ipaddr.o` in the object list and the `mconfig` file to give `build_ipaddr` a value, like the rest of the extensions. Please read the `INSTALL` file on how to correctly configure and compile `Xtables-addons`.

In the `mbuild` file in the top-level directory, you define whether or not to build a given extension, much like the kernel’s `.config`:

```
build_ipaddr=m
```

Then you also need the actual Makefile logic, which is also modeled upon the kernel’s build system. Add to the `extensions/Kbuild` file:

```
obj- $\{build\_ipaddr\}$  += xt_ipaddr.o
```

And to `extensions/Mbuild`:

```
obj- $\{build\_ipaddr\}$  += libxt_ipaddr.so
```

2.9.2 Standalone package

If you are writing your module in a out-of-tree standalone package, you can use a simple boilerplate Makefile:

```
# -*- Makefile -*-
MODULES_DIR := /lib/modules/$(shell uname -r)
KERNEL_DIR  := ${MODULES_DIR}/build

obj-m += xt_ipaddr.o

all:
    make -C ${KERNEL_DIR} M=$$PWD;
modules:
    make -C ${KERNEL_DIR} M=$$PWD $@;
modules_install:
    make -C ${KERNEL_DIR} M=$$PWD $@;
clean:
    make -C ${KERNEL_DIR} M=$$PWD $@;
```

Besides the Makefile, you need (of course) the source files and a kernel source tree. Calling ‘make’ then is everything needed to build `xt_ipaddr.ko`. You may pass `KERNEL_DIR=/path/to/builddir` to `make` in case you want to build against a kernel other than the one currently running.

The drawback compared to using `Xtables-addons` is of course that you do not get the pleasure to use the pre-existing glue code without doing some work yourself (such as copying it, keeping it up-to-date, etc.).

2.9.3 In-tree modifications to the kernel

The `xt_ipaddr.c` file should be put into `net/netfilter/` and `xt_ipaddr.h` into `include/linux/netfilter/`. Then you edit `net/netfilter/Makefile` and add a rule for your match to be built:

```
obj-$(CONFIG_NETFILTER_XT_MATCH_IPADDR) += xt_ipaddr.o
```

Finally, add the config option and a help text itself in `net/netfilter/Kconfig`. Where exactly you place this block in the `Kconfig` file does not matter, but we like to keep the list sorted, so `ipaddr` would currently find its place between the `NETFILTER_XT_MATCH_HELPER` and `NETFILTER_XT_MATCH_IPRANGE` config options.

```
config NETFILTER_XT_MATCH_IPADDR
    tristate '"ipaddr" source/destination address match'
    depends on NETFILTER_XTABLES
    ---help---
    The xt_ipaddr module allows you to match on source and/or
    destination address, and serves demonstration purposes only.
```

Please have a look at or `Git(author?)` [Git, GitJB], `Quilt(author?)` [QuiltFM, QuiltAG] or `StGit(author?)` [StGit] if you intend on submitting patches for your new module.

2.10 Summary

In this second part, we covered the basics of the Xtables module infrastructure and how to register our module with the framework by using a specific structure and functions. We discussed how to match a specific situation according to our idea, and how to go about IPv6 support, as well as a short section on how to get the module built.

3 Userspace plugin

The purpose of an iptables extension is basically to interact with the user. It will handle the arguments the user wants the kernel part to take into consideration.

3.1 Structural definition

`struct xtables_match` defines the vtable for one address family of a match extension. It is available from `<xtables.h>`.

```
struct xtables_match {
    const char *version;
    const char *name;
    uint8_t revision;
    uint16_t family;

    size_t size;
    size_t userspace_size;

    void (*help)(void);
    void (*init)(struct xt_entry_match *match);
    int (*parse)(int c, char **argv, int invert, unsigned int *flags,
                 const void *entry, struct xt_entry_match **match);
    void (*final_check)(unsigned int flags);
    void (*print)(const void *entry,
                  const struct xt_entry_match *match,
                  int numeric);
    void (*save)(const void *entry,
                 const struct xt_entry_match *match);
    const struct option *extra_opts;
};
```

3.2 Extension initialization

```
static struct xtables_match ipaddr_mt4_reg = {
    .version      = XTABLES_VERSION,
```

`version` is always initialized to `XTABLES_VERSION`. This is to avoid loading old modules in `/usr/libexec/iptables` with a newer, potentially incompatible iptables version.

`name` specifies the name of the module (obviously). It has to match the name set in the kernel module. Together with the next two fields, the `<name, revision, address family>` tuple is used to uniquely lookup the corresponding kernel module. `revision` specifies that this `xtables_match` is only to be used with the same-revision kernel-side Xtables match. `family` denotes what family this match operates on, in this case IPv4 (`NFPROTO_IPV4`), or IPv6 (`NFPROTO_IPV6`). You can also use `NFPROTO_UNSPEC`, which acts as a wildcard.

```
.name          = "ipaddr",
.revision      = 0,
.family        = NFPROTO_IPV4,
```

`size` specifies the size of our private structure in total. `userspace_size` specifies the part of the structure that is relevant to rule matching when replacing or deleting rules. It does

not apply to index-based deletion such as ‘`iptables -D INPUT 1`’, but for match/mask-based deletion, as in ‘`iptables -D INPUT -m rateest --rateest1 name1 --rateest2 name2 ... -j ACCEPT`’. When such a command is issued, the binary representation of that rule is constructed, and if it matches the binary blob from the kernel, it will be deleted. However, the `est1` and `est2` fields of `struct xt_rateest_mtinfo` are kernel-private fields and should be exempted from comparison. This is realized by specifying a `userspace_size` that is smaller than `size`, using `offsetof`¹. This is why kernel fields also should be at the end of the structure.

Usually, both `size` and `userspace_size` are the same, but there are exceptions like the aforementioned `xt_rateest` where the kernel module keeps additional information for itself. When `userspace_size` is less than `size`, it must not use `XT_ALIGN(offsetof(...))`, but just `offsetof(...)`.

```
.size          = XT_ALIGN(sizeof(struct xt_ipaddr_mtinfo)),
.userspace_size = XT_ALIGN(sizeof(struct xt_ipaddr_mtinfo)),
```

`help` is called whenever a user enters ‘`iptables -m module -h`’. `parse` is called when you enter a new rule; its duty is to validate the arguments. `print` is invoked by ‘`iptables -L`’ to show previously inserted rules.

```
.help          = ipaddr_mt_help,
.init          = ipaddr_mt_init,
.parse         = ipaddr_mt4_parse,
.final_check   = ipaddr_mt_check,
.print         = ipaddr_mt4_print,
.save          = ipaddr_mt4_save,
.extra_opts    = ipaddr_mt_opts,
};
```

It is possible to omit the `init`, `final_check`, `print`, `save` and `extra_opts` members (same as explicitly initializing them to `NULL`). `help` and `parse` must be defined.

The reason we use `ipaddr_mt4` sometimes and `ipaddr_mt` is because some functions and structures can be shared between the IPv4 and the IPv6 code parts, as we will see later. What exactly can be shared is tightly bound to the extension you are writing.

Each library must register to the running `iptables` (or `ip6tables`) program by calling `xtables_register_match`. The `_init` function is called when the module is loaded by `iptables`. For more information about it, see `dlopen(3)`. As a tiny implementation detail, note that `_init` is actually defined as a macro for `iptables`, and the keyword will be replaced by appropriate logic to wire it up with `iptables`, as we cannot strictly use `_init`, because the Glibc CRT (common runtime) stubs that will be linked into shared libraries, already do.

```
void _init(void)
{
    xtables_register_match(&ipaddr_mt_reg);
}
```

When `iptables` is built, this will expand to:

```
void __attribute__((constructor)) libxt_ipaddr_init(void)
```

¹See `libxt_rateest.c` in the `iptables` source package for an example.

so you may not use the name `libxt_ipaddr_init` for other functions, or you will get an unfortunate compile error.

In case you use the Xtables-addons framework, just directly write

```
static void _init(void)
```

i.e. with the `static` keyword and without the extra prototype above it, because modules are always compiled as shared library objects (`.so`) in Xtables-addons, so no symbols need to be globally visible.

3.3 Dumping rules – save function

If we have a ruleset that we want to save, iptables provides the tool `iptables-save` which dumps all your rules. It needs your extension’s help to interpret `struct xt_ipaddr_mtinfo`’s contents and dump proper rules. The output that is to be produced must be options as can be passed to iptables.

```
static void ipaddr_mt4_save(const void *entry,
    const struct xt_entry_match *match)
{
    const struct xt_ipaddr_mtinfo *info = (const void *)match->data;
```

We print out the source address if it is part of the rule.

```
    if (info->flags & XT_IPADDR_SRC) {
        if (info->flags & XT_IPADDR_SRC_INV)
            printf("! ");
        printf("--ipsrc %s ",
            xtables_ipaddr_to_numeric(&info->src.in));
    }
```

Note that `xtables_ipaddr_to_numeric` uses a static buffer, so you may not call it more than once before having the result printed out. It will convert a `struct in_addr` to numeric representation (“dotted notation”), e.g. 192.0.2.137. Then, we also print out the destination address if it is part of the rule.

```
    if (info->flags & XT_IPADDR_DST) {
        if (info->flags & XT_IPADDR_DST_INV)
            printf("! ");
        printf("--ipdst %s ",
            xtables_ipaddr_to_numeric(&info->dst.in));
    }
}
```

Note that output from the `save` function shall always be numeric, i.e. no IP addresses may be transformed to hostnames!

3.4 Status display – print function

In the same philosophy as the previous one, this function aims to print information about the rule, but in a freeform fashion. It is called by ‘`iptables -L`’, and you are free to output whatever you want, and how you want.

```
static void ipaddr_mt4_print(const void *entry,
    const struct xt_entry_match *match, int numeric)
{
    const struct xt_ipaddr_mtinfo *info = (const void *)match->data;

    if (info->flags & XT_IPADDR_SRC) {
        printf("src IP ");
        if (info->flags & XT_IPADDR_SRC_INV)
            printf("! ");
        if (numeric)
            printf("%s ", numeric ?
                xtables_ipaddr_to_numeric(&info->src.in) :
                xtables_ipaddr_to_anyname(&info->src.in));
    }

    if (info->flags & XT_IPADDR_DST) {
        printf("dst IP ");
        if (info->flags & XT_IPADDR_DST_INV)
            printf("! ");
        printf("%s ", numeric ?
            xtables_ipaddr_to_numeric(&info->dst.in) :
            xtables_ipaddr_to_anyname(&info->dst.in));
    }
}
```

Here, we use `xtables_ipaddr_to_anyname` in the `!numeric` case, to print a hostname when possible. The `numeric` case is triggered by the `-n` argument to `iptables` (‘`iptables -nL`’), which instructs `iptables` to not do DNS lookups.

3.5 Option parsing – parse function

This is the most important function because here, we verify if arguments are used correctly and set information we will share with the kernel part. It is called each time an option is found, so if the user provides two options, it will be called twice with the argument code provided in the variable `c`. The argument code for a specific option is set in the option table (see below).

```
static int ipaddr_mt4_parse(int c, char **argv, int invert,
    unsigned int *flags, const void *entry,
    struct xt_entry_match **match)
{
```

The `match` pointer is passed to a couple of functions so we can work on the same data structure. Once the rule is loaded, the data that is pointed to will be copied to kernel-space. This way, the kernel module knows what the user asks to analyze (and that is the point, is it not?).

```

struct xt_ipaddr_mtinfo *info = (void *)(*match)->data;
struct in_addr *addrs, mask;
unsigned int naddrs;

```

The cast is needed here since `data` is of type `char *`, rather than `void *`. Each command-line option, like `--srcip`, is assigned an integer value, stored in `c` here, to allow for specific actions to be done according to the inputted arguments. We will see later in this text how we map arguments to values.

```

switch (c) {

```

First, we check if the argument has been used more than once. If it appears to be the case, we call `xtables_error`, which will print the supplied error message and exit the program immediately with the status flag `PARAMETER_PROBLEM`. Else, we set `flags` and `info->flags` to the `XT_IPADDR_SRC` value defined in our header's file, to tell the kernel module that we want to do something. We will see our header file later.

Although both `flags` and `info->flags` seem to have the same purpose, but they really do not. The scope of `flags` is only this function (and the final check function), while `info->flags` is a field part of our structure which will be shared with the kernel.

```

case '1': /* --ipsrc */
    if (*flags & XT_IPADDR_SRC)
        xtables_error(PARAMETER_PROBLEM, "xt_ipaddr: "
                        "Only use \"--ipsrc\" once!");
    *flags |= XT_IPADDR_SRC;
    info->flags |= XT_IPADDR_SRC;

```

We verify whether the invert flag, `'!`', has been used on the command line (e.g. `'iptables -m ipaddr ! --ipsrc 192.168.0.137'`) and then set appropriate information in `info->flags`. There are a number of functions that take an IPv4/v6 address or hostname and turn it into a 32/128-bit entity. Here, we will use `xtables_ipparse_any`, which can take either a hostname or IP address, and will write the result to `addr` and `mask`. The `addrs` argument is used to store the addresses a host resolution might yield.

```

    if (invert)
        info->flags |= XT_IPADDR_SRC_INV;
    xtables_ipparse_any(optarg, &addrs, &mask, &naddrs);
    if (naddrs != 1)
        xtables_error(PARAMETER_PROBLEM,
                        "%s does not resolves to exactly "
                        "one address", optarg);
    /* Copy the single address */
    memcpy(&info->src.in, addrs, sizeof(*addrs));
    return true;

```

For demonstrational purposes, we will use `xtables_numeric_to_ipaddr` instead for the destination address. It transforms exactly one IPv4 address from dotted notation into a 32-bit entity:

```

case '2': /* --ipdst */
    if (*flags & XT_IPADDR_DST)
        xtables_error(PARAMETER_PROBLEM, "xt_ipaddr: "

```

```

                                "Only use \"--ipdst\" once!");
*flags |= XT_IPADDR_DST;
info->flags |= XT_IPADDR_DST;
if (invert)
    info->flags |= XT_IPADDR_DST_INV;
addrs = xtables_numeric_to_ipaddr(optarg, &info->dst.in);
if (addrs == NULL)
    xtables_error(PARAMETER_PROBLEM,
                  "Parse error at %s\n", optarg);
memcpy(&info->dst.in, addrs, sizeof(*addrs));
return true;
}
return false;
}

```

Every time an option was recognized, the parse function should return **true**, and **false** otherwise. This is because the parse function is also passed options that potentially belong to other modules, and if our function returns **false**, other parse functions are probed whether they recognize the option. In essence, every time you load a new match with iptables's **-m name** option, the option table for that specific match is added to the top of the option table search list.

3.6 Option validation – check function

This function is sort of a last chance for sanity checks. It is called when the user enters a new rule, after argument parsing is done and **flags** is filled with whatever values you chose to assign to it in your parse function.

```

static void ipaddr_mt_check(unsigned int flags)
{
    if (flags == 0)
        xtables_error(PARAMETER_PROBLEM, "xt_ipaddr: You need to "
                      "specify at least \"--ipsrc\" or \"--ipdst\".");
}

```

It is generally used to ensure that a minimum set of options or flags have been specified. Flags that conflict with one another, including an option with itself — in other words, specifying an option twice — is usually handled at the earliest point possible, in the parse function. But there are option combinations for which only the final check function makes sense to test them, as parse cannot “look forward”.

3.7 Options structure

Earlier, we discussed that every option is mapped to a single argument code value. The **struct option** is the way to achieve it. For more information about this structure, I strongly suggest you read `getopt(3)`. You need to include `<getopt.h>` for it.

```

static const struct option ipaddr_mt_opts[] = {
    {.name = "ipsrc", .has_arg = true, .val = '1'},
    {.name = "ipdst", .has_arg = true, .val = '2'},
    {NULL},
};

```

3.8 Rule initialization – init function

The `init` function can be used to populate our `xt_ipaddr_mtinfo` structure with defaults before `parse` is called. If you do not need it, just omit initialization of the `init` field in our `ipaddr_mt_reg` structure (like it was done above).

```
static void ipaddr_mt_init(struct xt_entry_match *match)
{
    struct xt_ipaddr_mtinfo *info = (void *)match->data;

    inet_pton(PF_INET, "192.0.2.137", &info->dst.in);
}
```

In this example, the default destination addresses is set to 192.0.2.137, and unless the user overrides it with `--ipdst`, this address will be used. (Actually, the destination address will not be tested in the `xt_ipaddr` kernel module unless `--ipdst` is given, so this example is sort of a no-op.) The initialization is often not needed because the memory pointed to by `match->data` is already zeroed so that iptables extensions do not need to take care of clearing `match->data` before being able to use it in the `parse` function.

3.9 Short usage text – help function

This function is called by `'iptables -m match_name -h'`. It should give an overview of the available options and a very brief short description. Everything that is longer than one line should be put into the manpage (see section 3.11).

```
static void ipaddr_mt_help(void)
{
    printf(
        "ipaddr match options:\n"
        "[!] --ipsrc addr    Match source address of packet\n"
        "[!] --ipdst addr     Match destination address of packet\n"
    );
}
```

3.10 IPv6 support

Similarly to the kernel module, you will also want to add IPv6 support in the iptables extension. For that, we need a separate `struct xtables_match`.

```
static struct xtables_match ipaddr_mt6_reg = {
    .version      = XTABLES_VERSION,
    .name         = "ipaddr",
    .revision     = 0,
    .family       = NFPROTO_IPV6,
    .size         = XT_ALIGN(sizeof(struct xt_ipaddr_mtinfo)),
    .userspacesize = XT_ALIGN(sizeof(struct xt_ipaddr_mtinfo)),
    .help         = ipaddr_mt_help,
    .parse        = ipaddr_mt6_parse,
    .final_check  = ipaddr_mt_check,
    .save         = ipaddr_mt6_opts,
    .print        = ipaddr_mt6_print,
```



```

        .opts          = ipaddr_mt_opts,
};

```

As mentioned earlier, a few functions can be shared, such as `ipaddr_mt_help` or `ipaddr_mt_check`, because they are independent of the address family used. For the others, we need IPv6-specific parse, save and print functions that handle IPv6 addresses:

```

static int ipaddr_mt6_parse(int c, char **argv, int invert,
    unsigned int *flags, const void *entry,
    struct xt_entry_match **match)
{
    struct xt_ipaddr_mtinfo *info = (void *)(*match)->data;
    struct in6_addr *addrs;

    switch (c) {
    case '1': /* --ipsrc */
        if (*flags & XT_IPADDR_SRC)
            xtables_error(PARAMETER_PROBLEM, "xt_ipaddr: "
                "Only use \"--ipsrc\" once!");
        *flags |= XT_IPADDR_SRC;
        info->flags |= XT_IPADDR_SRC;
        if (invert)
            info->flags |= XT_IPADDR_SRC_INV;
        addrs = xtables_numeric_to_ip6addr(optarg);
        if (addrs == NULL)
            xtables_error(PARAMETER_PROBLEM, "xt_ipaddr: "
                "Parse error at %s", optarg);
        memcpy(&info->src.in6, addrs, sizeof(*addrs));
        return true;
    }
    return false;
}

```

I have left out the case '2', you can surely add it yourself (it is in the `libxt_ipaddr.c` file in the Xtables-addons git repository anyway). The only interesting change here is that we use `xtables_numeric_to_ip6addr`, and the appropriate `in6_addr` structures this function takes (`->src.in6`, `->dst.in6`). You should also be able to write the save and print functions; all that is needed is `xtables_ip6addr_to_numeric` and `xtables_ip6addr_to_anyname`, respectively. Add registering the `ipaddr_mt6_reg` structure to `_init`, and you are done:

```

static void _init(void)
{
    xtables_register_match(&ipaddr_mt4_reg);
    xtables_register_match(&ipaddr_mt6_reg);
}

```

3.11 Documentation

The `help` function should only give a really short overview of the available options. Some iptables extensions already have so many options — yet the minimum amount of necessary help text — that it fills a screenful. Please take the time to write anything else that you want

to make the user aware of into a separate manpage file. When `iptables` is built, the manpage files are merged into one, to complete `iptables.8` and `ip6tables.8`. The build process will create a subsection for the module, so we do not need to. The man text could be:

```
The ipaddr module matches on source and/or destination IP address.
.TP
[\fB!\fP] \fB--ipsrc\fP \fIaddr\fP
Match packets that have \fIaddr\fP as source address.
.TP
[\fB!\fP] \fB--ipdst\fP \fIaddr\fP
Match packets that have \fIaddr\fP as destination address.
.PP
The ipaddr module serves only as a demonstration. It is equivalent to
the iptables \fB-s\fP and \fB-d\fP options, but ipaddr does not support
masks.
```

Granted, our module is simple, and so is the manpage. (It also serves as an introduction to write nroff markup.) When you build `iptables` and look at the completed manpage afterwards, using ‘`man -l iptables.8`’ perhaps or a viewer of your choice, you can see that `\fB` is for bold, `\fI` for italic and `\fP` for normal². `.TP` will do an indentation appropriate for option and description, and `.PP` will return to the default paragraph indentation.

3.12 Building the extension

3.12.1 Using the Xtables-addons package

Place the modules’ files — `xt_ipaddr.c` and `xt_ipaddr.h` — into the `extensions/` directory and modify the `Kbuild` file to include `xt_ipaddr.o` in the object list and the `mconfig` file to give `build_ipaddr` a value, like the rest of the extensions. Please read the `INSTALL` file on how to correctly configure and compile `xtables-addons`.

Place the extension module `libxt_ipaddr.c` into the `extensions/` directory and modify the `Mbuild` file to include `libxt_ipaddr.so` in the object list and the `mconfig` file to give `build_ipaddr` a value, if you have not done so yet.

mconfig:

```
build_ipaddr=m
```

extensions/Mbuild:

```
obj- $\{$ build_ipaddr $\}$  += libxt_ipaddr.so
```

Please read the `INSTALL` file on how to correctly configure and compile `xtables-addons`.

To make use of the module without copying it to the `xtables` module directory, you will have to use something like:

```
XTABLES_LIBDIR=$PWD:/usr/libexec/xtables iptables -A INPUT -m ipaddr ...;
```

when you are inside the `extensions` directory. The `XTABLES_LIBDIR` environment variable, if set, instructs `iptables` to search for extensions in the given directories. You want to make sure the original directory — `/usr/libexec/xtables` here, but it might be different on your system or if you had just previously built `iptables` in your own home directory.

²It actually means “previous” and acts like `` or `</i>` does in HTML for a preceding `` and `<i>`, respectively.

3.12.2 Standalone package

To compile the iptables extension, all you need is the development header files from iptables (usually in a package called `iptables-devel`) and some means to turn `libxt_ipaddr.c` into a shared library object, `libxt_ipaddr.so`. You can use a Makefile such as:

```
CFLAGS = -O2 -Wall
lib%.so: lib%.o
    gcc -shared -o $@ $^;
lib%.o: lib%.c
    gcc ${CFLAGS} -D_INIT=lib$_init -c -o $@ $<;
```

and then call ‘`make libxt_ipaddr.so`’, or wire up more Makefile logic to automatically build the targets.

3.12.3 In-tree modifications to the iptables package

The filename for the extension source code should be `libxt_ipaddr.c` and be put into the `extensions/` directory. There is no need to edit a Makefile, as it will automatically glob up all files that match `libxt_*.c`. Now build iptables. To enable debugging, you can override the default `CFLAGS` with the debug flag. `-ggdb3` includes lots of debug, in the preferred format and with GDB extensions (= all that you could ever need). It is also highly recommended to pass in `-O0` to turn off optimization (an “O” followed by a zero) and therefore instruction reordering, otherwise gdb will jump around source lines, making debugging hard.

```
./configure CFLAGS="-ggdb3 -O0";
```

iptables has recently moved to autotools, so uses configure. It also does not require a kernel source tree (anymore). Please read the `INSTALL` file to find out more!

To test your extension without having to install iptables to a system location, in other words, to run it from the build directory, set the `--with-xtlibdir` option:

```
./configure --with-xtlibdir="$PWD/extensions";
```

then you can test `ipaddr`:

```
./iptables -m ipaddr -h;
./iptables -A INPUT -m ipaddr --ipsrc 192.0.2.137;
./ip6tables -A INPUT -m ipaddr --ipsrc 2001:db8::1302;
```

To see if it is working, check either the `printk` messages that accumulated in the kernel log, or use ‘`iptables -vL`’ to watch the counters increasing. Make sure that either `xt_ipaddr.ko` can be loaded by `modprobe` or is already loaded.

3.13 Summary

In this part, we discussed the purpose of the iptables extension module. We covered the internals of each function and how the main structure `xt_ipaddr_mtinfo` is used to keep information that will be copied to the kernel side for further consideration. We also looked at the iptables structure and how to register our new extension.

4 Tricks and traps

4.1 The packet buffer

`struct sk_buff` is an essential structure throughout networking — it carries your data; data to match on, or to modify.

The life of an `skb` begins when the kernel, specifically a network driver, reads a packet from the network card's buffer into RAM³. At this point, all you essentially have is a pointer to said buffer in `skb->data`, and its length in `skb->len`. `skb->data` is actually a pointer that will move as the packet is passed on to the upper layers. In case of an Ethernet driver, the driver will call `eth_type_trans`, which will reset the MAC header pointer by calling `skb_reset_mac_header` on the `skb`⁴. The function does no more than copy (the value of) `skb->data` to `skb->mac_header` (implementation might vary). `eth_type_trans` will then advance `skb->data` by the size of the Ethernet header and decrease `skb->len` by the same amount using the `skb_pull` function, so that `skb->data` will be pointing to the start of the layer-3 header and `skb->len` contains the remaining length. At this point it is guaranteed that the layer-2 header is complete and that the memory block pointed to by the pointer that is returned by `skb_mac_header` is accessible for up to `skb->mac_len` bytes. Afterwards, the `skb` is handed to the generic receive routine, `netif_receive_skb` or any of the likes.

`netif_receive_skb` resets the network and transport header pointers, using `skb_reset_network_header` and `skb_reset_transport_header`, respectively. Packets will now⁵ be sent to ingress shaping, Ebtables if a bridge device was involved, macvlan devices, and finally the layer-3 protocol handler⁶. As packets have not yet been processed by layer 4 yet, you cannot rely on the `skb_transport_header` function to return a meaningful value. For IPv4 match extensions, you can use `struct xt_match_param->thoff` that `ip_tables/xt_ip` filled in. For IPv6 match extensions, this field will only be filled when the rule contains a protocol specified with the `ip6tables -p` flag, e.g. “`-p tcp`”. In all other cases, `ip_hdrlen` or `ipv6_skip_exthdr` has to be called manually. The reason Netfilter/Xtables does not set the transport header pointer in the `skb` or always provide the transport header offset is likely to be due to performance considerations.

The layer-3 header is validated by the corresponding layer-3 protocol handler so that you can always safely use `ipv6_hdr` in an extension that is registered for the `NFPROTO_IPV6` family, and `ip_hdr` for `NFPROTO_IPV4`, respectively. Because packets are passed to Ebtables first before they go to any layer-3 handlers, `NFPROTO_BRIDGE` extensions cannot use the `ip_hdr` and/or `ipv6_hdr` functions directly, but must use the safe boundary-checking function `skb_header_pointer` to obtain any data. `NFPROTO_UNSPEC` extensions generally fall under the same rule, because they are valid for all families/protocols, however, you can examine `struct xt_match_param->family` and/or `struct xt_target_param->family` and check which Xtables family invoked the match/target function. If it is `NFPROTO_IPV4/NFPROTO_IPV6` you can use `ip_hdr/ipv6_hdr`, too.

In the output path, things are done in reverse. An `skb` of appropriate size is allocated and filled with your data. `skb->data` as such will point to the data that is queued for being sent out, and `skb->len` will contain its length. The TCP send function will then call `skb_push` to make room for the TCP header in front of `skb->data`. It does so by decreasing `skb->data` and increasing `skb->len`. and calling `skb_reset_transport_header`, so that `skb->transport_header` will point to what `skb->data` currently points to. This is repeated for all the lower layers; once it is the IP layer's turn, `skb_push` is used again to stack the layer-3

³For an example, see `drivers/net/niu.c`, function `niu_process_rx_pkt`.

⁴See `net/ethernet/eth.c`, function `eth_type_trans`.

⁵See `net/core/dev.c`, function `netif_receive_skb`.

⁶See `net/ipv6/ip6_input.c`, function `ipv6_rcv`.

header onto it and to then reset the network header to the data pointer, etc. Note that the transport header pointer is set by the time Xtables is called in the output path, so you may take this shortcut if useful.

4.2 Registering multiple structures at once

As we have seen earlier in section 2.8, trying to register multiple structures at once can become a tedious job with regard to the error path. Xtables provides four convenient functions to (un)register arrays of matches and targets. When applied to our code, the `ipaddr_mt_reg` structure and init and exit functions now look like this:

```
static struct xt_match ipaddr_mt_reg[] __read_mostly = {
    {
        .name      = "ipaddr",
        .revision   = 0,
        .family     = NFPROTO_IPV4,
        .match      = ipaddr_mt4,
        .matchsize  = sizeof(struct xt_ipaddr_mtinfo),
        .me         = THIS_MODULE,
    },
    {
        .name      = "ipaddr",
        .revision   = 0,
        .family     = NFPROTO_IPV6,
        .match      = ipaddr_mt6,
        .matchsize  = sizeof(struct xt_ipaddr_mtinfo),
        .me         = THIS_MODULE,
    },
};

static int __init ipaddr_mt_init(void)
{
    return xt_register_matches(ipaddr_mt_reg,
        ARRAY_SIZE(ipaddr_mt_reg));
}

static void __exit ipaddr_mt_exit(void)
{
    xt_unregister_matches(ipaddr_mt_reg, ARRAY_SIZE(ipaddr_mt_reg));
}
```

4.3 Using connection tracking modules

Sometimes you want to operate on connections rather than packets. For that to be successful, packets must actually be inspected by the connection tracking code — essentially making Netfilter stateful. Xtables extensions that require connection tracking will try to load it as needed. One way this can happen is due to symbol dependencies, i.e. a named function or variable is needed. All of the IPv4 modules that do stateful NAT will generally make use of the `nf_nat_setup_info` symbol from `nf_conntrack.ko`. The dependencies between kernel

modules are computed at compile time⁷. `modprobe`⁸ adheres to the “depends-on” names listed in a compiled kernel module and loads the dependencies first, for example `nf_conntrack.ko` before `ipt_MASQUERADE.ko`. Such dependencies are essential. The `MASQUERADE` code just cannot run without `nf_nat_setup_info`, so a failure to load `nf_conntrack.ko` results in a failure to load `ipt_MASQUERADE.ko`.

Then there are run-time dependencies. It does not make much sense to load IPv6 connection tracking if you never use IPv6, so it is preferable to not load it when not needed. But once required, it will be requested by the kernel, which in turn calls the `modprobe` userspace binary itself. Run-time dependencies are allowed to fail to resolve, and code using such either tries something else or aborts gracefully.

```
static bool conntrack_mt_check(const struct xt_mtchk_param *par)
{
    return nf_ct_l3proto_try_module_get(par->family) == 0;
}
```

This is the very quick way how to do it. Once a rule that uses the `conntrack` match is inserted, it will load the appropriate layer-3 connection tracking module, because without, it will not be possible to get the connection structure (`struct nf_conn`) for a particular packet in the main match function — the `nf_ct_get` function that is used to obtain the associated connection for a packet just returns `NULL` and the whole match never matches.

Connection tracking is split into multiple modules and categories. First of all we have the core, `nf_conntrack`, which actually includes all the layer-4 trackers. Then there are currently two layer-3 trackers, `nf_conntrack_ipv4` and `nf_conntrack_ipv6`. Lastly there are layer-5 trackers, such as `nf_conntrack_irc`.

`nf_ct_l3proto_module_try_get` tries to load the module appropriate for the address family used, the latter of which actually depends on whether you tried to insert an IPv4 `ip_tables` rule or an IPv6 `ip6_tables` rule and on the `xt_match` structure (all of this equally applies to targets) that was invoked as part of it. It will also increase the reference count of the layer-3 protocol module so that it cannot be removed using `rmmmod` while the `ip_tables` rule is in place. Only after all rules depending on connection tracking (`ct`) have been removed, the `ct` module may be removed too. It is therefore important to drop the reference count once a rule is removed:

```
static void conntrack_mt_destroy(const struct xt_mtdtor_param *par)
{
    nf_ct_l3proto_module_put(par->family);
}
```

The following `lsmod` excerpt indicates that connection tracking is in use. I have two rules that use the `conntrack` match, so that accounts for two references to `xt_conntrack` and two references to `nf_conntrack_ipv4`. The other two references to `nf_conntrack_ipv4` come from `iptable_nat`⁹ and `nf_nat`¹⁰.

Module	Used by
--------	---------

⁷Actually, link time — it is when `modpost` runs. I chose the “compile-time” wording because it certainly is not at runtime during `modprobe`. (And neither at `depmod` time.)

⁸`insmod` does not do automatically load dependencies and thus often “fails” because symbols from dependent modules are not yet registered in the kernel.

⁹It is listed after all — has a symbol dependency.

¹⁰It seems to irregularly grab `nf_conntrack_ipv4` however.

```

iptables_nat      1
nf_nat            3 ipt_REDIRECT, ipt_MASQUERADE, iptable_nat
xt_conntrack      2
nf_conntrack_ipv4 4 iptable_nat
nf_conntrack      5 ipt_MASQUERADE, iptable_nat, nf_nat,
                  xt_conntrack, nf_conntrack_ipv4

```

4.4 Alignment of extension data

The kernel as well as userspace can run in various execution environments and combinations. You can have a 32-bit kernel with a 32-bit userspace, a 64-bit kernel with a 64-bit userspace and a 64-bit kernel with a 32-bit userspace. For all these cases, data needs to be interchangeable. When a rule is transferred from or to the kernel, it is serialized into a contiguous binary stream. The structures are sent as they appear in memory, so that the meaning of the binary blob is actually dependent on the remote side's interpretation. It is therefore important that both userspace and kernel use the same struct definitions on the blob, and actually even a definition that has the same binary representation in both worlds. Types like `long` can have different sizes in different environments, hence the use of fixed types like `__u32` is mandatory.

In a struct, types are by default only aligned as much as required by the type and environment. Looking at most used hardware, i.e. x86, x86_64, and some others like sparc64, a `__u16` must be 2-aligned (must start at an address divisible by 2), and `__u32` must be 4-aligned. Things get interesting with `__u64` — in a 64-bit environment, it must be 8-aligned, while on 32-bit, 4-alignment suffices because a 32-bit environment does not process 64-bit quantities in one operation, but instead splits it up into multiple 32-bit operations. GCC 4.x uses 8-alignment for 64-bit quantities even on x86, but not so on some ARM configurations, and CRIS is said to only have a 1-alignment requirement on any type. Without telling the compiler to use a different alignment, it will pack members in a moderate way, at least adhering to the alignment specifications for the configured environment. GCC does not reorder, so there can still be holes. The Netfilter “communication protocol” however follows the 64-bit model, and 64-bit quantities must always be 8-aligned, even in 32-bit mode. To do so, all affected members must be tagged with `__attribute__((aligned(8)))`. To facilitate this, `aligned_u64` is a shorthand macro for `__u64 __attribute__((aligned(8)))` (similarly for `aligned_le64` and `aligned_be64`):

```

struct foo {
    __u8 id;
    aligned_u64 count;
    __u32 bar;
};

```

The serialized bytestream consists of concatenations of various structures: `struct xt_entry_match`, `struct ip6t_ip6`, followed by a group for each match consisting of `struct ip6t_entry` and the private match structure. Each of these structs is supposed to be 8-aligned, so they need to be padded where necessary. To this end, the `XT_ALIGN` macro should be used which rounds up the value passed in up to the next boundary.

```

.matchsize = XT_ALIGN(sizeof(struct foo)),

```

4.5 Attaching kernel-specific data

Generally, the shared structure, `xt_ipaddr_mtinfo` in our case, only contains the necessary parameters needed to drive the match. However, there are times when the kernel module itself

needs to do bookkeeping. `xt_quota` for example keeps track of the number of bytes that passed the match, on a per-match basis. To achieve this, it adds a few extra fields to the structure (`<linux/netfilter/xt_quota.h>`):

```
struct xt_quota_mtinfo {
    __u32 flags;

    /* Used internally by the kernel */
    aligned_u64 quota;
    struct xt_quota_mtinfo *master __attribute__((aligned(8)));
};
```

The first kernel-only variable should be aligned to the size of the largest representation it can attain, which is 64-bit (8 bytes) for pointers. To do so, you use the `aligned` attribute as shown.

When the kernel-private data gets too big, you can use an indirection instead, and allocate state when the rule is inserted (and free when it is deleted). Consider this hypothetical `xt_bigipaddr` match that records the timestamps of the eight most recent processed packets¹¹:

```
struct xt_bigipaddr_state {
    __u32 seen[8];
};

struct xt_bigipaddr_mtinfo {
    __u16 match_flags, invert_flags;

    /* Used internally by the kernel */
    struct xt_bigipaddr_state *state __attribute__((aligned(8)));
};

static bool xt_bigipaddr_check(const struct xt_mtchk_param *par)
{
    struct xt_bigipaddr_mtinfo *info = par->matchinfo;

    info->state = kmalloc(sizeof(*info->state), GFP_KERNEL);
    if (info->state == NULL)
        return false;
    return true;
}

static void xt_bigipaddr_destroy(const struct xt_mtdtor_param *par)
{
    struct xt_bigipaddr_mtinfo *info = par->matchinfo;
    kfree(info->state);
}
```

Because the kernel data is internal and can change, it should not be used for comparing rule equality in userspace. The `userspace_size` field in the iptables userspace module must be set to the actual portion that is the “key” (if we were to use SQL terminology). With the appropriate offsetof, only the first two members are compared, which is what we want.

¹¹We could have also directly written `union nf_inet_addr *seen`, but only the clever C programmers should think about that.


```

struct xtables_match foo = {
    .userspace_size = offsetof(struct xt_bigipaddr_mtinfo, state);
};

```

Now `xt_quota`, which served as an example too, is a bit of a more special case, as the `quota` member is not only in the private data, but also directly in `struct xt_quota_mtinfo`. This is because the userspace module wants to print the when the user runs `iptables -L` or similar. If the field was hidden behind a pointer, userspace could not access it, because kernel pointers are invalid in userspace — and it is only possible to do so-called “shallow copies”¹². With this hack come the issues of updating values on SMP.

4.6 SMP problems

You might have noticed the ominous `master` field in the `struct xt_quota_mtinfo`. It has to do with the way Xtables stores rulesets in memory. After the `check` function has run (successfully), Xtables will duplicate the entire rule (including `struct xt_quota_mtinfo`) for every CPU core, for NUMA optimization reasons(**author?**) [QuotaOnSMP]. This obviously creates a difficult decision: which `struct xt_quota_mtinfo` to update?

In the `check` function, a separate memory area is allocated which will hold the quota value and which is decoupled from `struct xt_quota_info`.

```

struct xt_quota_info *q = par->matchinfo;
q->master = kmalloc(sizeof(*q->master), GFP_KERNEL);

```

Now when `matchinfo` is duplicated, the duplicates’ addresses may change, but the `info->master` member remains unchanged in all copies. It is then easy to just update the master’s counters from all CPU cores:

```

struct xt_quota_mtinfo *q = par->matchinfo;
q->master->quota -= skb->len;

```

This alone does not solve the problem — mentioned in the previous reference — that Xtables will copy the wrong struct to userspace (e.g. for `iptables -vL`). The best approximation for this is to copy the shared quota value to the per-cpu variable everything the match function is called.

```

/* Copy quota back to matchinfo so that iptables can display it */
q->quota = q->priv->quota;

```

To get at the real value that is currently held in memory at `q->priv->quota`, other mechanisms need to be used. The `xt_quota2` module from Xtables-addons for example exports the exact quota through `procs`¹³.

4.7 Deferred rule deletion on table replacement

When tables are replaced, the new rules are loaded into the kernel first before the old ones are removed. This also means that the `checkentry` function is called before `destroy`. This is important to know when information is shared between two rules, for example lists of IP addresses in `xt_recent` or geoip lists in `xt_geoip` — when an existing rule is changed with

¹²Compare with “deep copies”, where pointers are followed.

¹³Though, this is actually done so, because `quota2` also allows the quota to be set while the rule is active.

iptables-restore, the already populated address list is not cleared/changed since the reference count never dropped to zero.

Calling `iptables` manually — this will do *two* table replacements — may clear the address list, but *only* if there was exactly one reference to “foo” in all tables:

```
iptables -D ... -m recent --name foo;
iptables -A ... -m recent --name foo;
```

4.8 A bit of coding style

Do not needlessly define your own incarnation of a debug macro. Instead, use the existing `pr_devel` which is enabled once `DEBUG` is enabled.

```
/* Avoid this */
#if 1
#    define DEBUGP printk
#else
#    define DEBUGP(format, args...)
#endif
DEBUGP("Hello World\n");
/* Use this */
#if 1
#    define DEBUG 1
#endif
pr_devel("Hello World\n");
```

“`#if 1`” may also be replaced with “`#ifdef CONFIG_SOME_OPTION`”, in case a Kconfig option is used to disable or enable debug info.

5 Target extensions

Targets can be really versatile. Common categories and examples are:

- mangling the packet payload – DSCP, TCPMSS, HL/TTL
- setting up NAT mappings – MASQUERADE, NETMAP, REDIRECT
- replying to packets (original packet is not modified) – REJECT
- changing skb, packet or connection parameters – CLASSIFY, CONNMARK, MARK, NO-TRACK, TRACE
- just watching packets, e.g. for statistical or analytical purposes (most often, matches are used instead) – LOG/NFLOG, RATEEST
- moving the packet to userspace – NFLOG, NFQUEUE
- other actions – SYSRQ

A few snippets from existing target modules will be explained in this chapter to demonstrate how they interact with Xtables.

Focus is on the `xt_ECHO` sample target for explanation of the skeletal structure. `xt_ECHO` which will return all bytes that have been written to a port — in effect, this is the “echo” protocol as defined in (**author?**) [RFC862]. It will be limited to UDP, because implementing a TCP engine is a somewhat bigger task and would extend beyond the scope of this document.

5.1 Naming convention

Just like for matches (see section 2.4), there is also a convention for targets. All it takes is replacing the `_mt` part by `_tg`. While targets’ names are upper-case, symbols will remain lower-case.

- `echo_tg_reg` – structure/object containing target metadata and vtable
- `echo_tg` (or `echo_tg4`, `echo_tg6` when it uses distinct functions) – the target (“action”) function
- `echo_tg_check` – function to check for validity of parameters in our struct
- `echo_tg_destroy` – function when rule is deleted
- `struct xt_echo_tginfo` and `struct xt_echo_tginfoN` – structure for our own data (for revision *N*)

5.2 Structural definition

This is the `xt_target` structure, excluding internal fields. It is also defined in `<linux/netfilter/x_tables.h>`.

```
struct xt_target_param {
    const struct net_device *in, *out;
    const struct xt_target *target;
    const void *targinfo;
    unsigned int hooknum;
```

```

        uint8_t family;
};

struct xt_tgchk_param {
    const char *table;
    const void *entryinfo;
    const struct xt_target *target;
    void *targinfo;
    unsigned int hook_mask;
    uint8_t family;
};

struct xt_tgdtor_param {
    const struct xt_target *target;
    void *targinfo;
    uint8_t family;
};

struct xt_target {
    const char name[XT_FUNCTION_MAXNAMELEN-1];
    uint8_t revision;
    unsigned short family;
    const char *table;
    unsigned int hooks;
    unsigned short proto;

    unsigned int targetsizes;
    unsigned int (*target)(struct sk_buff *skb,
                           const struct xt_target_param *par);
    bool (*checkentry)(const struct xt_tgchk_param *par);
    void (*destroy)(const struct xt_tgdtor_param *par);

    struct module *me;
};

```

Xtables-addons uses a slightly different `target` function signature to cope with kernels before 2.6.24, so if you plan on writing your module with the help of the Xtables-addons glue code, do not be surprised of an extra compiler warning if copying code verbatim. The signature is:

```

unsigned int (*target)(struct sk_buff **pskb,
                      const struct xt_target_param *par);

```

5.3 Module initialization

The structure looks quite the same as matches do (see section 2.3), so the initialization is straightforward:

```

static struct xt_target echo_tg_reg __read_mostly = {
    .name      = "ECHO",
    .revision  = 0,
    .family    = NFPROTO_IPV4,
    .proto     = IPPROTO_UDP,

```

The target shall be limited to the `filter` table, because it does not make too much sense in `mangle`, and is totally out of place in `nat` (for IPv4) which is only called on the first packet of a (pseudo-)connection. `hooks` is a bitmask and may contain zero or more of the following flags:

- 1 << NF_INET_PRE_ROUTING
- 1 << NF_INET_INPUT
- 1 << NF_INET_FORWARD
- 1 << NF_INET_OUTPUT
- 1 << NF_INET_POST_ROUTING

Kernels before 2.6.25(-rc1) used `NF_IP_` and `NF_IP6_` prefixes, but because the values are the same, they have been collapsed into `NF_INET_`. Note that `arptables` and `ebtables` use their own hooks. If `hooks` is not set, it is initialized to 0 by default, which implies that this target can be used in all chains.

The target shall further only be usable from `INPUT`, `FORWARD` and `OUTPUT`, although this is already guaranteed by restricting it to the filter table, which has only these three chains. It is therefore optional, but shown here.

```

        .filter      = "table",
        .hooks       = (1 << NF_INET_LOCAL_IN) |
                       (1 << NF_INET_FOWARD) |
                       (1 << NF_INET_LOCAL_OUT),
        .target       = echo_tg4,
        .targetsize   = XT_ALIGN(0),
        .me           = THIS_MODULE,
};

```

A `targetsize` of zero is used here, because we decided not to add any options right now. Hence there is also no private data structure and no header file to define, a `checkentry` function is also absent since there is nothing to validate nor a layer-3 tracking module needs to be loaded in this implementation.

The rest is, again, known standard code for (un)register the target on module insertion/removal and some metadata:

```

static int __init echo_tg_init(void)
{
    return xtables_register_target(&echo_tg_reg);
}

static void __exit echo_tg_exit(void)
{
    xtables_unregister_target(&echo_tg_reg);
}

MODULE_DESCRIPTION("Xtables: RFC 862 \"echo\" protocol implementation");
MODULE_LICENSE("GPL");
MODULE_ALIAS("ipt_ECHO");
MODULE_ALIAS("ip6t_ECHO");

```

Xtables also provides “plural” functions for target (un)registration for your convenience that take an array of `struct xt_targets`; they are called `xtables_register_targets` and `xtables_unregister_targets`.

5.4 Verdicts

Each rule can be assigned a target, which can be seen as an “action” that is to be done. It is only called when all matches assigned with a rule have matched. It is invoked with ‘`iptables ... -j ECHO`’ for example, and the three special targets `ACCEPT`, `DROP` and `RETURN` that directly map to a verdict.

On a code base, target extensions still need to return a verdict on their behalf. Depending on the nature of the target, either `NF_ACCEPT` or `NF_DROP` is chosen for terminating targets, while `XT_CONTINUE` is used for targets that do not cause rule traversal to stop.

Possible verdict return values for the function are:

- `XT_CONTINUE` – continue with next rule. Most commonly used by “watcher” (`ip6t_LOG`, `xt_NFLOG`) and mangling targets, to allow for multiple mangling transformations.
- `NF_DROP` – stop traversal in the current table hook and drop packet. It is the standard action of any target that has somehow processed the (original) packet (`ip6t_REJECT`).
- `NF_ACCEPT` – stop traversal in the current table hook and accept packet. Used by targets that set up a NAT mapping.
- `XT_RETURN` – return to the previous chain or default chain policy. This is an internal target only, no modules use it to avoid creating confusion on behalf of the user.

For security reasons, packets that cannot be processed due to a memory allocation failure, routing problem, or any other problem should be discarded with `NF_DROP` so that they will not leak from, that is, bypass, the firewall.

5.5 Replying with packets

A warning beforehand: sending packets from within Netfilter causes re-entrancy. `ip_tables`, `ip6_tables`, etc. store the jump stack for a table within that (per-cpu) table itself¹⁴. Once control from the second invocation returns to the original target, the jump stack will have been overwritten, and `XT_CONTINUE` or `XT_RETURN` — so-called “relative verdicts” — cannot be used without causing undefined behavior. An absolute verdict from the `NF_*` group must be returned.

Prior to 2.6.24, the target function uses an indirect `skb` pointer, `const struct sk_buff **pskb`; for those kernels, use `*pskb` when accessing the `skb`¹⁵. Xtables-addons continues to provide an API with a `pskb` for being able to interface with older kernels.

There will be a lot of local variables in this function. When writing real targets, it is advised to split big functions up.

```
static unsigned int echo_tg4(struct sk_buff *skb,
    const struct xt_target_param *par)
{
    const struct udphdr *oldudp;
    const struct iphdr *oldip;
    struct udphdr *newudp, oldudp_buf;
    struct iphdr *newip;
    struct sk_buff *newskb;
    unsigned int addr_type, data_len, offset;
    void *payload;
```

¹⁴A block of heap memory associated with the per-cpu table is used in a stack fashion.

¹⁵Again, no `#ifdef LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 25)` trickery is needed when you use the Xtables-addons package.

The incoming skb might have IP options which we do not want in the outgoing skb/packet, therefore the new skb is rebuilt from scratch. (For TCP this is even more important because there can also be TCP options besides IP options.) A possible alternative is to use the `skb_copy` (or `skb_copy_expand`) function instead and somehow strip all potentially unwanted pieces.

At first, pointers to the IPv4 and UDP headers are obtained. `ip_hdrlen` is a function that returns the size of the IPv4 header including options. The `offset` variable will be used during the course of this function as the index to data that has yet to be copied to the new packet. The third argument to `skb_header_pointer` specifies the amount of bytes to obtain starting at `offset`. Not always do all fields need to be retrieved. It would be perfectly valid to use 6 instead of `sizeof(struct udphdr)` in the second `skb_header_pointer` call, because we are not really interested in the UDP checksum. If the region to be extracted from the packet is out-of-bounds, NULL will be returned. This provides a check for maliciously short packets, and such should be dropped right away. Also, any UDP packet without any payload is ignored too.

Our simplistic implementation will linearize skbs to keep the code as simple as possible. (If the skb is already linear, nothing happens.)

```

    if (skb_linearize(oldskb) < 0)
        return NF_DROP;

    oldip = ip_hdr(oldskb);
    offset = ip_hdrlen(oldskb);
    oldudp = skb_header_pointer(oldskb, par->thoff,
                                sizeof(struct udphdr), &oldudp_buf);
    if (oldudp == NULL)
        return NF_DROP;
    if (ntohs(oldudp->len) <= sizeof(struct udphdr))
        return NF_DROP;

    offset += sizeof(struct udphdr);

```

`ip_hdr` (and other accessors, like `ipv6_hdr`) can be used without problems since the layer-3 header (without IPv4 options, though) is always linear, i. e. it is one continuous stream of bytes and is not fragmented or split across multiple skbs¹⁶. On the other hand, `skb_header_pointer` needs to be used for data that could potentially be non-linear. `skb_header_pointer` will, if a skb boundary is crossed, copy data from multiple skbs into the buffer pointed to by its fourth argument, making the desired bytes available in linear memory. If the byte range that should be extracted is already linear, a pointer is returned, making the operation cheap.

In the next step, the new skb is allocated. The fact that it should have the same size as the original packet (minus IP options) should be obvious. Some extra space for the link layer processing seems needed, so `LL_MAX_HEADER` is added¹⁷. The `length` field of `struct udphdr` includes the UDP header's own size, so `sizeof(struct udphdr)` is not added again. Do not forget to use the `ntohs` and `ntohl` functions when dealing with fields in network packets — only predefined functions such as `ip_hdrlen` for example do the endianness conversion already. The allocation must be done using `GFP_ATOMIC` because the target function might be run in interrupt context, where sleeping is not allowed and a failure to get hold of memory should result in an immediate return from the allocator with a NULL result.

```

newskb = alloc_skb(LL_MAX_HEADER + sizeof(struct iphdr) +

```

¹⁶This is done in `net/ipv4/ip_input.c`, function `ip_rcv`

¹⁷The exact use for `LL_MAX_HEADER` is beyond the scope of this document. For now, it is best to look at existing target extensions within the kernel and copy their behavior.

```

        ntohs(oldudp->len), GFP_ATOMIC);
if (nskb == NULL)
    return NF_DROP;

skb_reserve(nskb, LL_MAX_HEADER);
skb_reset_network_header(nskb);

```

What follows is filling out the fields of a newly attached `iphdr` structure. `skb_put` extends the skb's tail pointer by as many bytes as specified and returns the original pointer (of type `unsigned char *`, which is why a cast is needed to get it to the desired `struct iphdr *` without warnings).

```

newip = (void *)skb_put(newskb, sizeof(struct iphdr));
newip->version = 4;
newip->ihl     = sizeof(struct iphdr) / 4;
newip->tos     = oldip->tos;
newip->id      = oldip->id;
newip->frag_off = oldip->frag_off;
newip->protocol = oldip->protocol;
newip->check    = 0;
newip->saddr    = oldip->daddr;
newip->daddr    = oldip->saddr;

```

The code sets the fragment identification number and offset to the original values, which should make fragment processing work, since the packet length is not changed. It is not always that easy however, and extensions are free to ignore fragments¹⁸. On many systems, the packet defragmenter is enabled because a number of modules, such as the IPv4 NAT engine and `xt_conntrack` depend on the connection tracking subsystem being loaded, so Xtables often does not get to see fragments anyway.

```

newudp = (void *)skb_put(nskb, sizeof(struct udphdr));
newudp->source = oldudp->dest;
newudp->dest   = oldudp->source;
newudp->len    = oldudp->len;
newudp->check  = 0;

```

The UDP header is equipped with a zero checksum here, which means that we did not care about integrity and no checksum was calculated, which is just the right thing for testing. In modules with real-world applicable use however, you should properly calculate checksums.

Now comes the copy operation. Because the presence of non-linear skbs has been ruled out above by linearizing it, a few shortcuts can be taken: `NULL` can be passed in as the fourth argument to `skb_header_pointer`, and `memcpy` can be used. There are probably critical opinions about linearizing skbs just to make the code simpler. Extracting pieces of the oldskb with `skb_header_pointer` is likely just as expensive; in the typical coded case, it always takes up the requested amount of stack memory, whereas `skb_linearize` instead causes a heap allocation in the event of a non-linear skb. Stack usage vs. a potential allocation, that is the trade-off.

```

data_len = htons(oldudp->len) - sizeof(*oldudp);
payload  = skb_header_pointer(oldskb, ip_hdrlen(oldskb) +
                               sizeof(*oldudp), data_len, NULL);
memcpy(skb_put(newskb, data_len), payload, data_len);

```

¹⁸At the potential cost of an unhappy user who avoids having `nf_defrag` loaded.

After the packet has been constructed, it is still necessary to select an output route for it and actually send it off. It is a fair bit of boilerplate code that is unfortunately duplicated in a few places in the kernel without having being grouped in a function so far. It is left out here, but can be looked up in the `xt_ECHO` code in the Xtables-addons package. There is still something left, however. The new skb gets the same conntrack entry as the oldskb, so it will be part of the connection that is seen in Netfilter.

```

    nf_ct_attach(newskb, oldskb);
    ip_local_out(newskb);
    return NF_DROP;
}

```

If the packet is handled by our module, it must not be passed to the real UDP stack which would otherwise be called after all the Netfilter hooks completed. If it were to be let through to the UDP core, an ICMP error might be generated because there is no open socket on that port, or an application that does have a socket open sends some sort of unwanted negative reply. This must be avoided, so `NF_DROP` is used as the final verdict.

```

# iptables -I INPUT -i lo -p udp --dport 2345 -j ECHO;
# tcpdump -Xs0 -lni lo udp &
# echo "xtables-addons" | socat - udp-sendto:localhost:2345
03:35:42.732411 IP 127.0.0.1.32847 > 127.0.0.1.2345: UDP, length 15
    0x0000:  4500 002b 0cf7 4000 4011 2fc9 7f00 0001  E...@.@./.....
    0x0010:  7f00 0001 804f 0929 0017 fe2a 7874 6162  .....0.)...*xtab
    0x0020:  6c65 732d 6164 646f 6e73 0a                les-addons.
03:35:42.732468 IP 127.0.0.1.2345 > 127.0.0.1.32847: UDP, length 15
    0x0000:  4500 002b 0000 4000 4011 3cc0 7f00 0001  E...@.@.<.....
    0x0010:  7f00 0001 0929 804f 0017 0000 7874 6162  .....).0....xtab
    0x0020:  6c65 732d 6164 646f 6e73 0a                les-addons.

```

5.6 Changing packet payload

This section needs expansion.

The packet payload can simply be changed by toying around with the skb. You can write to `skb->data`, or even resize it if need be. What you do need to pay attention to is that you possibly need to regenerate layer-4 (e.g. TCP/UDP) and layer-3 (IPv4, but not in IPv6) checksums.

```

static unsigned int memfry_tg(struct sk_buff *skb,
    const struct xt_target_param *par)
{
    struct udphdr *udph;
    unsigned char *data;
    unsigned int data_len;

    if (!skb_make_writable(skb, skb->len))
        return NF_DROP;
}

```

`skb_make_writable` will ensure here that we will have exclusive ownership of the packet. It may also, as a matter of fact, cause the pointers inside the skb to change, so if you have copied `skb->data` to any of your variables, you will have to re-fetch the `skb->data` pointer afterwards.

```

    udp      = skb->data + ip_hdrlen(skb);
    data     = skb->data + ip_hdrlen(skb) + sizeof(struct udphdr);
    data_len = skb->len - ip_hdrlen(skb) - sizeof(struct udphdr);

    for (i = 0; i < data_len; ++i)
        data[i] ^= i;
    /* checksumming? */

    return XT_CONTINUE;
}

```

You can take shortcuts in calculating the checksum if you can assure that the checksum remains the same after you applied your transformation. This is for example the case when you use XOR, but I am not too much into the maths, so this is left up to you.

5.7 Checksumming

Would someone like to contribute to this section?

```

#include <linux/checksum.h>

void csum_replace2(__sum16 *sum, __be16 from, __be16 to);
void csum_replace4(__sum16 *sum, __be32 from, __be32 to);
void inet_proto_csum_replace2(__sum16 *sum, struct sk_buff *skb,
    __be16 from, __be16 to, bool pseudohdr);
void inet_proto_csum_replace4(__sum16 *sum, struct sk_buff *skb,
    __be32 from, __be32 to, bool pseudohdr);

```

Setting the checksum to 0 or 0xFFFF has special meaning... setting `skb->ip_summed` indicates something different again even.

5.8 Modifying packet parameters

One of the “easier” classes of targets are those that do not play much with the packet payload, in fact, that only do read operations on it. Xtables has quite a number of them — `xt_MARK` and `xt_CONNMARK`, just to name two. These influence the `skb` or connection parameters and otherwise do not do very much besides being very flexible about what they do, providing masked bit operations on parameters. `xt_IPMARK` is not much different, albeit simple enough to show what it does:

```

static unsigned int ipmark_tg4(struct sk_buff *skb,
    const struct xt_target_param *par)
{
    const struct xt_ipmark_tginfo *info = par->targinfo;
    const struct iphdr *iph = ip_hdr(skb);
    uint32_t mark;

    if (info->sel == XT_IPMARK_SRC)
        mark = ntohl(iph->saddr);
    else
        mark = ntohl(iph->daddr);
}

```

```

    mark >>= info->shift;
    mark &= info->andmask;
    mark |= info->ormask;
    skb->mark = mark;
    return XT_CONTINUE;
}

```

5.9 Setting up a NAT mapping

Another common target scenario is setting up a NAT mapping for a connection. To do this, a range of addresses and/or ports that the NAT engine may use must be handed to `nf_nat_setup_info`. Additionally, the target is only valid in the `nat` table. `struct nf_nat_range` and the `IP_NAT_*` defines are available through `<net/netfilter/nf_nat.h>`.

```

static unsigned int marksnat_tg(struct sk_buff *skb,
    const struct xt_target_param *par)
{
    const struct iphdr *iph = ip_hdr(skb);
    struct nf_nat_range range = {};
    struct nf_conn *ct;

    /* for debug */
    ct = nf_ct_get(skb, &ctinfo);
    NF_CT_ASSERT(ct != NULL && (ctinfo == IP_CT_NEW ||
        ctinfo == IP_CT_RELATED));
}

```

There are currently three range flags available. `IP_NAT_RANGE_MAP_IPS` advises the NAT subsystem to do translation on the layer-3 addresses, `IP_NAT_RANGE_PROTO_SPECIFIED` makes sure that the given layer-4 protocol port range in `range.min` and `range.max` are used, and `IP_NAT_RANGE_PROTO_RANDOM`, with which `range.min` and `range.max` are ignored and instead, the NAT engine selects a randomly-chosen port number at runtime.

```

range.flags    = IP_NAT_RANGE_MAP_IPS;
range.min_ip   = ntohl(iph->saddr) & 0FFFFFF00;
range.min_ip  |= skb->mark & 0xFF;
range.min_ip   = htonl(range.min_ip);
range.max_ip   = range.min_ip;

```

Shown here is a sample mapping dependent upon the Netfilter mark, using the lower 8 bits of the mark for constructing the new (source) address, and the next 8 and 8 bits for the port range in some way that the range is between [1024, 65535].

```

    if (iph->protocol == IPPROTO_TCP ||
        iph->protocol == IPPROTO_UDP) {
        range.flags |= IP_NAT_RANGE_PROTO_SPECIFIED;
        range.min.tcp.port = (skb->mark & 0x00FF00) << 2;
        range.max.tcp.port = (skb->mark & 0xFF0000) >> 8;
    }

    return nf_nat_setup_info(ct, &range, IP_NAT_MANIP_DST);
}

```

Destination NAT is furthermore only valid in the PREROUTING and OUTPUT hooks while Source NAT is only in the POSTROUTING and OUTPUT chains. This has to be enforced by setting the `table` and `hooks` members of the registration structure accordingly.

```
static struct xt_target marksnat_tg_reg __read_mostly = {
    .name      = "MARKSNAT",
    .revision   = 0,
    .family    = NFPROTO_IPV4,
    .table     = "nat",
    .hooks     = (1 << NF_INET_POST_ROUTING),
    .target    = marksnat_tg,
    .targetsize = XT_ALIGN(0),
    .me        = THIS_MODULE,
};
```

5.10 Rule validation – checkentry function

Like with Xtables matches, the `checkentry` function is called whenever a rule is about to be inserted and allows for checks to be done and run-time dependencies to be loaded, as discussed in section 2.6. Like before, the `checkentry` function may be omitted.

The `xt_TCPMSS` kernel module provides an example of how `checkentry`-based hook verification is done. Here, if the user manually sets the MSS, nothing special will happen. But when automatically setting the MSS relative to the PMTU, we need the PMTU value, which is only available after the routing decision, so one can only use this method to set the MSS from the FORWARD, OUTPUT and POSTROUTING chains, when an output route has been decided for the packet.

```
static int tcpmss_tg_checkentry(const struct xt_tgchk_param *par)
{
    const struct tcpmss_tg *info = par->targinfo;

    if (info->mss == XT_TCPMSS_CLAMP_PMTU &&
        (hook_mask & ~((1 << NF_INET_FORWARD) |
            (1 << NF_INET_LOCAL_OUT) |
            (1 << NF_INET_POST_ROUTING)))) != 0)
        return -EINVAL;
    return 0;
}
```

5.11 Rule destruction – destroy function

As with matches, targets can have a `destroy` function as a counterpart to `checkentry`. It may be omitted, too.

```
static void xyz_tg_destroy(const struct xt_tgdtor *par)
{
    pr_info("A \"%s\" target was destroyed.\n",
        par->target->name);
}
```

5.12 Notes for in-tree modifications

If you depend on a certain table like `mangle`, `nat` or `raw`, you should add a dependency line in the Kconfig file for your target. For some reason, this is not done for the `filter` table; anyway:

```
config NETFILTER_XT_TARGET_CONNMARK
    tristate '"CONNMARK" target support'
    depends on IP_NF_MANGLE || IP_NF6_MANGLE
```

Tables are still per-family (i. e. not generic enough to be handled in `x_tables.c`), which is why there are two symbols to depend on (`IP_NF_MANGLE` (`CONFIG_IP_NF_MANGLE`) and `IP_NF6_MANGLE`). Other symbols are `IP_NF_RAW` and `IP_NF6_RAW` for the `raw` table, and `NF_NAT` for the (IPv4) `nat` table. IPv6 does not have a `nat` table.

Part II

Connection Tracking

There are no specific prerequisites. The API has remained pretty much the same over at least 2.6.23 to 2.6.25. There have been slight type changes for 2.6.26(-rc1) (and which will be used here). Xtables-addons does not provide API compatibility coverage for connection tracking yet as there are no modules merged that would require such, but its build system might still be very handy in writing a module.

6 nf_conn structure

Sometimes it is necessary to retrieve connection parameters. The `nf_ct_get` function will find the connection associated with a packet, if there is such, and return it along with the connection status. For `nf_ct_get`, you need to include `<net/netfilter/nf_conntrack.h>`, and for `enum ip_conntrack_info`, you will need `<linux/netfilter/nf_conntrack_common.h>`. The reason this is split is that the `linux/` directory contains headers which are exported to userspace, and the values for constants like `IP_CT_NEW` are surely useful in userspace too, while `nf_ct_get` is a function only available in the kernel.

```
#include <linux/netfilter/nf_conntrack_common.h>

enum ip_conntrack_info ctinfo;
struct nf_conn *ct;

ct = nf_ct_get(skb, &ctinfo);
```

There exist a multitude of connection states and statuses, and you can match all of them with the `conntrack` match in Xtables, or print the connection information with `xt_LOGMARK`, a target extension in the Xtables-addons package. Note that the connection tracking subsystem is invoked after the `raw` table has been processed, but before the `mangle` table.

```
if (ct == NULL)
    pr_info("This is --ctstate INVALID\n");
```

`ct` can be `NULL` if the packet has been declared `INVALID` by the connection tracking subsystem. This can happen for example if a TCP SYN is sent on an already-existing connection.

```
else if (ct == &nf_conntrack_untracked)
    pr_info("This one is not tracked\n");
```

The `NOTRACK` target can be used (in the `raw` table) to exempt a packet from connection tracking; this is especially useful when using the `TARPIT` target(**author?**) [Chaostables]. It can also be used on any packet you would like to drop, but generally, people do not bother because it often incurs a rule duplication. Just dropping packets in the `filter` table means that a connection entry will remain until it times out, which generally works well enough. Default timeout depends on protocol and implementation, and is usually between 30 seconds to 2 minutes when you drop connections marked as `NEW`. Speaking of `NEW`:

```

else if (ctinfo % IP_CT_IS_REPLY == IP_CT_NEW)
    pr_info("This is the first packet in a connection\n");
else if (ctinfo % IP_CT_IS_REPLY == IP_CT_RELATED)
    pr_info("Welcome Mr. Bond, we have been expecting you\n");
else if (ctinfo % IP_CT_IS_REPLY == IP_CT_ESTABLISHED)
    pr_info("You can figure out this one!\n");

```

If `ct` is not `NULL` and not the fake connection tracking entry used for “untracking” packets, the connection is a valid one and its state can be found in `ctinfo`. `enum ip_conntrack_ctinfo` combines both the connection state and the packet direction with an arithmetic add instead of a flag, which may seem a bit confusing. Here goes:

- `IP_CT_NEW` – new connection created by this packet
- `IP_CT_RELATED` – this packet starts a new but expected connection
- `IP_CT_ESTABLISHED` – connection is established, packet is in “original” direction
- `IP_CT_ESTABLISHED + IP_CT_IS_REPLY` – connection is established, packet is in “reply” direction
- `IP_CT_RELATED + IP_CT_IS_REPLY` – expected new connection started, and packet is in the “reply” direction. It may be surprising how the first packet in a connection can be in the reply direction (note: reply direction of the *expected* connection not the original one). This is actually used for ICMP replies, at which point `RELATED+REPLY` seems logical.
- `IP_CT_NEW + IP_CT_IS_REPLY` is not used and not valid.

By using `ctinfo % IP_CT_IS_REPLY` (in this case analogous to `ctinfo & ~IP_CT_IS_REPLY` if a flag would had been used), the connection state can be extracted. The direction in which the packet flows could be extracted using `ctinfo / IP_CT_IS_REPLY`, but the convenience macro `CTINFO2DIR(ctinfo)`, as defined in `<linux/netfilter/nf_conntrack_tuple_common.h>` uses an open-ended range comparison instead¹⁹.

¹⁹Something whose implementation details should be quickly forgotten again and the macro just be used. Too bad this mess cannot be easily changed as it is exported to userspace.

Connection trackers

Connection trackers are one of the essential parts of the connection tracking infrastructure, and related to that, stateful firewalling. Their job is to associate an IP packet with a connection and to assure the correctness of packets and their parameters. TCP streams for example are inspected for proper window size and correct state transitions.

In its simplest case, the source and destination addresses are copied from the packet to a `struct nf_conntrack_tuple`, the latter of which is then chained along with other tuples, forming the table of known connections.

Connection tracking is split up into two categories, layer-3 and layer-4 modules, allowing maximum modularity. There are also layer-5 trackers though they are referred to as “connection helpers” because their existence does not effect the original connection, but future connections.

7 Layer-3 connection tracker

7.1 Objective

Of course, the question came up what geeky idea this chapter could be filled, and the results were disenchanting.

Showcasing a sample layer-3 connection tracker that is not one of those included in the kernel is going to be a very tough job. Not only because IPv4 and IPv6 are the most predominant protocols used, but also because the kernel does not currently have Netfilter hooks for anything besides these two plus a few special ones.

This led to some quite interesting undertakings. Jan came up with a connection tracker for IPX, but reviving old DOS games in virtual machines turned out to be a longer-term task due to technical problems with modern operating systems. Resorting to an ARP connection tracker was not too fruitful either after recognizing that there were no Netfilter hooks in the ARP input/output paths. So that idea was also scrapped because we would like to avoid touching the kernel and putting the reader through a perhaps long recompile and installation cycle, being not only outside the scope of this book but also way beyond networking.

7.2 Structural definition

The structure for layer-3 trackers is defined in `<net/netfilter/nf_conntrack_l3proto.h>`. It contains packet-to-tuple association, tuple inversion functions and one to obtain the layer-4 protocol number.

```
struct nf_conntrack_l3proto {
    const char *name;
    uint16_t l3proto;

    bool (*pkt_to_tuple)(const struct sk_buff *skb, unsigned int nhoff,
                        struct nf_conntrack_tuple *tuple);
    bool (*invert_tuple)(struct nf_conntrack_tuple *inverse,
                        const struct nf_conntrack_tuple *orig);
    int (*print_tuple)(struct seq_file *,
                      const struct nf_conntrack_tuple *);

    int (*get_l4proto)(const struct sk_buff *skb, unsigned int nhoff,
                     unsigned int *dataoff, uint8_t *protonum);
}
```



```

    struct module *me;

};

```

Layer-3 connection trackers are pretty useless without an actual layer-4 part. The `pkt_to_tuple` and `invert_tuple` functions will be called, but since that is all the `nf_conntrack_l3proto` structure makes available, you will not be seeing any entries in the Netfilter connection table, observable by running `conntrack -L` or looking at the kernel-provided map at `/proc/net/nf_conntrack`. Only when there is an appropriate layer-4 tracker registered, connection tracking will actually be done and events be generated that can be monitored using `conntrack -E`.

The `get_l4proto` function should inspect the packet and return the layer-4 protocol number from the `nexthdr` field (IPv6) or the `Protocol` field (IPv4). It may return `-NF_ACCEPT` if the connection is not to be tracked.

7.3 Generic L4 tracking

While there are only layer-4 trackers for the most common protocols (actually protocols where it makes sense to do so), there is a not insubstantial number of protocols that get tracked, as far as that is possible, using a generic tracker. AH and ESP are two that fall into this category, for example.

The generic tracker maps all packets for a layer-4 protocol to one connection, this is as good as the logic can get. The connection tracking table will then show an entry like:

```

# conntrack -L | grep "^unknown"
unknown  50 537 src=192.168.0.137 dst=192.168.16.34 packets=12 bytes=1456
src=192.168.16.34 dst=192.168.0.137 packets=12 bytes=2704 mark=0 use=1

```

We hope the reader will excuse the shortness of this chapter. Be assured that writing a layer-3 tracker, should the need arise, is much the same like a layer-4 tracker which will be covered right in the next chapter.

8 Layer-4 connection tracker

8.1 Structural definition

The struct for layer-4 trackers is contained in `<net/netfilter/nf_conntrack_l4proto.h>` and is as follows. The order in which the callbacks `error-destroy` are arranged here depict the order they are executed in as a packet flows through.

```
struct nf_conntrack_l4proto {
    const char *name;
    uint16_t l3proto;
    uint8_t l4proto;

    int (*error)(struct sk_buff *, unsigned int dataoff,
                 enum ip_conntrack_info *ctinfo, unsigned int pf,
                 unsigned int hooknum);
    bool (*pkt_to_tuple)(const struct sk_buff *skb,
                        unsigned int dataoff,
                        struct nf_conntrack_tuple *tuple);
    bool (*invert_tuple)(struct nf_conntrack_tuple *inverse,
                        const struct nf_conntrack_tuple *original);
    int (*packet)(struct nf_conn *ct, const struct sk_buff *skb,
                 unsigned int dataoff, enum ip_conntrack_info ctinfo,
                 unsigned int pf, unsigned int hooknum);
    bool (*new)(struct nf_conn *ct, const struct sk_buff *skb,
               unsigned int dataoff);
    void (*destroy)(struct nf_conn *ct);

    int (*print_conntrack)(struct seq_file *s,
                          const struct nf_conn *ct);
    int (*print_tuple)(struct seq_file *s,
                      const struct nf_conntrack_tuple *tuple);

    struct module *me;
};
```

It also has packet-to-tuple conversion/association and tuple inversion, but also “packet”, “new” and “destroy”. While these extra callbacks are not layer-4 specific, the `nf_conntrack_l3proto` structure does not contain them²⁰ due to lack of use.

8.2 Objective

In this chapter we have a look at an ESP connection tracking module. It works absolutely, but has no practical field value because the generic tracking handles the common daily usage with IPsec traffic just as well. If you need to track specific SPI streams however for some very obscure reason, this module is for you.

ESP is encrypted and hence there is no way for a non-endpoint to look inside it²¹. Even so, for “tunnel” mode and actually any sort of (unencrypted) tunnels/encapsulation, we often do

²⁰The unused callbacks in `struct nf_conntrack_l3proto` were removed for 2.6.26.

²¹Tunnel endpoints may use the “policy” match to inspect transformed connections.

not want to inspect the inner contents, because that is what a tunnel is about — to track the tunnel connection itself²².

8.3 Module initialization

The `name` member gives a short identifying string to `conntrack` which is used for reporting to userspace. It should be kept simple, and should definitely not have any spaces in it. `l3proto` and `l4proto` specify the layer-3 and layer-4 protocols for which this helper should be invoked, respectively. Note that in case of an IPv6 packet, the first non-extension header's protocol number or, if that did not exist, the last header's protocol number is used for matching with a loaded connection tracking module — it is therefore currently not possible²³ to invoke a helper based on e.g. the presence of an IPv6 Destination Options header.

```
static struct nf_conntrack_l4proto esp_ctrack_reg __read_mostly = {
    .name          = "esp",
    .l3proto       = NFPROTO_IPV6,
    .l4proto       = IPPROTO_ESP,
```

The rest of the struct consists of pointers to functions that make up the connection tracking helper module:

- `esp_ctrack_pkt2tuple` – mapping a packet to a (connection) tuple
- `esp_ctrack_new` – function that is called once a new “connection” (read: connection tracking entry) is added
- `esp_ctrack_packet` – packet processing function, e.g. updating the internal tracking state. TCP uses this to go from `SYN_SENT` to `SYN_RECV`, for example.
- `esp_ctrack_invtuple` – inverting a tuple
- `esp_ctrack_prct` – print connection tracking entry
- `esp_ctrack_pttuple` – print connection tuple

```
    .pkt_to_tuple   = esp_ctrack_pkt2tuple
    .new            = esp_ctrack_new
    .packet         = esp_ctrack_packet,
    .invert_tuple   = esp_ctrack_invtuple,
    .print_conntrack = esp_ctrack_prct,
    .print_tuple    = esp_ctrack_pttuple,
    .me             = THIS_MODULE,
};
```

8.4 The tuple structures

`union nf_conntrack_man_proto` contains the “manipulable protocol”²⁴ part, somehow a collective term for (one side of) the layer-4 specific tuple data such as ports.

²²Which does not mean that there would be no way to analyze in-tunnel traffic (unencrypted tunnels) with only stock Linux kernels.

²³In previous footnotes I always said “already developed”, now here is something that is yet to be done!

²⁴Well, this is what the abbreviation `man_proto` suggests to me, given that the data stored in it is can be modified as part of NAT.

```

union nf_conntrack_man_proto {
    __be16 all;

    struct {
        __be16 port;
    } tcp;
    ...
};

```

There will be a member called `all`; its exact type depends on how the developers laid `nf_conntrack` out for a particular kernel release. It is internal to `nf_conntrack` as an optimization to tuple matching and should rather not be used in any way by modules. However, since an external module might want to use the available space without patching any of the kernel files, some hackery is needed, and it comes with its limitations too.

```

struct esp_man_proto {
#ifdef sizeof(union nf_conntrack_man_proto) >= sizeof(uint32_t)
    __be32 spi;
#elif sizeof(union nf_conntrack_man_proto) >= sizeof(uint16_t)
    __be16 spi;
#endif
};

static inline struct esp_man_proto *
esp_priv(union nf_conntrack_man_proto *ptr)
{
    return (void *)ptr;
}

```

This does look indeed gory and ugly, but it allows us to use at least the 16 bits of space that `union nf_conntrack_man_proto` is big in 2.6.25. The compound is only 16 bits in size since all in-kernel connection trackers only need to track this much. Therefore, the sample ESP module needs to make a cut in the packet-to-tuple mapping correctness — the SPI truncation may cause packets to be falsely attributed to a connection, but I will assume that for any non-static key setup you will be using random SPI numbers generated by an IPsec/IKE daemon. For some future modules these 16 bits may not be enough and would warrant a change of `union nf_conntrack_man_proto` in the kernel source.

8.5 Packet to conntrack

When a packet enters the connection tracking subsystem, it is passed on to the appropriate layer-4 connection tracking module, to the `pkt_to_tuple` hook. This function should map the packet to a connection tuple, the latter of which uniquely identifies a “connection” in Netfilter terms.

```

static int esp_ctrack_pkt2tuple(const struct sk_buff *skb,
                               unsigned int dataoff, struct nf_conntrack_tuple *tuple)
{
    const struct ip_esp_hdr *ptr;
    struct esp_man_proto *man;
    struct ip_esp_hdr buf;

```

Since only the first four bytes of the ESP header are needed to obtain the SPI, passing in 4 for the third argument is sufficient; no need to retrieve the entire ESP header.

```

    ptr = skb_header_pointer(skb, dataoff, 4, &buf);
    if (ptr == NULL)
        return false;
    man = esp_priv(&tuple->dst.u);
    man->spi = ptr->spi;
    return true;
}

```

The connection tracking core zeroes the tuple before it is handed to the layer-3 and layer-4 connection trackers. When the `esp_ctrack_pkt2tuple` function receives it, it is guaranteed that `tuple->src.u` and `tuple->dst.u` (the layer-4 parts) are zeroed, and `tuple->src.u3` and `tuple->dst.u3` (the layer-3 parts) are filled in.

8.6 Tuple inversion

When a packet arrives, it gets compared to all pre-existing tuples to see if a packet is in the other direction of a connection.

The connection tracking engine gets fed packets without knowing what interface they came on in or which interface they will leave on, if any²⁵. Connection tracking does not even want to know that, usually, since policy routing may cause the packet to enter or leave on different interfaces each time, especially in setups with load balancing.

IPsec flows are unidirectional in nature, and they do not provide a way to find something that identifies the direction in the other direction. In TCP for example, each packet has both a source and a destination port. In an ESP packet however, you will only find a destination SPI (“port”). Therefore, tuple inversion is not possible for ESP and the function should return `false`.

```

static bool esp_ctrack_invtuple(struct nf_conntrack_tuple *inverse,
    const struct nf_conntrack_tuple *original)
{
    return false;
}

```

But in most cases where tuples are invertible, all it takes is to actually do the inversion. `orig` and `tuple` point to different memory locations, so no temporary copying to the stack is needed.

```

static bool tcp_invert_tuple(struct nf_conntrack_tuple *tuple,
    const struct nf_conntrack_tuple *orig)
{
    inverse->src.u.all = original->dst.u.all;
    inverse->dst.u.all = original->src.u.all;
}

```

²⁵This information can be retrieved from the `skb`, but that is an implementation detail of `sk_buff` and should be avoided if possible.

8.7 Dumping entries

There exist two hooks which are called for dumping information about the connection and the tuple. The first is for example used by the TCP tracker to return the state of the TCP connection (e. g. `SYN_SENT`), while the second is used to print per-tuple layer-4 parameters such as source and destination port, or SPI for ESP.

```
static int esp_ctrack_prct(struct seq_file *s, const struct nf_conn *ct)
{
    /* Nothing special to report */
    return 0;
}
```

The return value for the two functions shall be that of `seq_printf`, which returns the number of bytes written to the stream.

```
static int esp_ctrack_prtuple(struct seq_file *s,
    const struct nf_conntrack_tuple *tuple)
{
    unsigned int spi = ntohl(esp_priv(&tuple->dst.u));
    return seq_printf(s, "spi=0x%x ", spi);
}
```

8.8 Summary

To see that the module works, you can put the module on any host through which some ESP traffic flows. Trying to use this connection tracking module on a tunnel/transport endpoint will not work as intended, because the decapsulation takes place before Connection Tracking gets to see the skb, and encapsulation takes place after CT already saw the skb, though Xtables will see the skb twice — once decapsulated and once encapsulated.

So under the assumption that the ESP tracking module is put to a test on a host that is not an endpoint, one can observe the spawning of a new Netfilter connection once a ESP packet has been sighted:

```
# conntrack -E
[NEW] esp 50 120 src=2001:db8::31:1 dst=2001:db8::32:2 spi=0xdc0adc5e
[ASSURED] src=2001:db8::31:1 dst=2001:db8::32:2 sport=22 dport=39760
```

9 Connection tracking helper

This section is incomplete.

A connection tracking helper is a module that inspects packet flows, more generically, the payload of packets, and sets up connection tracking expectations. They can then be matched in the Xtables firewall with the `RELATED` (for the first packet) and `ESTABLISHED` (all further packets) state bits of the conntrack match. This is commonly seen in user scripts as

```
iptables -m conntrack --ctstate RELATED,ESTABLISHED
```

In this chapter, we will glimpse at a connection tracking helper that will look at HTTP streams and create expectations for further connections.

When the Netfilter connection tracking subsystem is loaded, you can depend on all packets entering the stack to be already defragmented.

9.1 Naming convention

- `http_cthelper_reg` – structure with metadata and vtable
- `http_cthelper` – the main helper function
- `http_cthelper_destroy` – a destroy function (if used)

9.2 Skeletal structure

The `struct nf_conntrack_helper` can be obtained by including `<net/netfilter/nf_conntrack_helper.h>`

```
struct nf_conntrack_helper {
    const char *name;
    unsigned int max_expected;
    unsigned int timeout;

    /* Tuple of connection to analyze */
    struct nf_conntrack_tuple tuple;

    /* Our helpful function */
    int (*help)(struct sk_buff *skb, unsigned int protoff,
                struct nf_conn *ct, enum ip_conntrack_info ctinfo);
    void (*destroy)(struct nf_conn *ct);

    struct module *me;
};
```

Besides internal fields that should not be meddled with (left them out in this document, of course), the structure contains function pointers for conversion from and to Netlink attributes, but that is currently outside the scope of this document.

9.3 Initialization

Though not quite visible directly, `struct nf_conntrack_helper` describes a connection tracker helper for exactly one tuple only, that is — in simple terms — one TCP/UDP port per `struct`. This may seem like a huge drawback given that protocols can generally be run over arbitrary

port numbers, but this is how helpers currently are implemented. As a workaround, connection tracking helpers define an array of `nf_conntrack_helpers` and the user gets to choose a fixed port list the administrator wants to have inspected.

```
static struct nf_conntrack_helper ftp_cthelper_reg __read_mostly = {
    .name          = "ftp",
    .max_expected  = 1,
    .timeout       = 60,
    .tuple         = {
        .src.l3num    = NFPROTO_IPV4,
        .dst.protonum  = IPPROTO_TCP,
        .dst.u.tcp.port = 21,
    },
    .help          = ike_cthelper,
    .me            = THIS_MODULE,
};
```

Of course, this code block is just an example; the `nf_conntrack_ftp` module does the registration a bit more elegant in that it fills out the structure at module initialization time instead of typing it out, well, because the user can change them.

Part III

Netfilter Core

10 Netfilter Hooks

The networking code has a number of calls that invoke Netfilter, and everything that is tied to it (given the modules are loaded). The most common hook components are Xtables (firewall), connection tracking, IPv4 NAT engine and the IPVS Virtual Server. `ip6_input.c` calls out to a specific hook:

```
return NF_HOOK(NFPROTO_IPV6, NF_INET_PRE_ROUTING, skb,
               dev, NULL, ip6_rcv_finish);
```

which will invoke all prerouting hooks. `NF_HOOK` is a monstrous macro crypt until it enters the real function, `nf_hook_slow`, and I leave that to the particular developer to munge that part on his own. (No worries, just training you.)

10.1 Skeletal structure

`struct nf_hook_ops` contains the vtable and metadata for a hook, like name and associated protocol. The definition can be obtained by including `<linux/netfilter.h>`. Hooks are per-protocol; one structure can only be registered for one protocol, so you need an array of `nf_hook_ops` if you plan to register multiple hooks.

```
struct nf_hook_ops {
    unsigned int pf;
    unsigned int priority;
    unsigned int hooknum;
    unsigned int (*hookfn)(unsigned int hooknum, struct sk_buff *skb,
                           const struct net_device *in,
                           const struct net_device *out,
                           int (*okfn)(struct sk_buff *));
    struct module *owner;
}
```

10.2 Initialization

The `pf` member associates the hook with the given group of calls. Possible values are listed in `netfilter.h`. The most common values are, of course, `NFPROTO_IPV6`, `NFPROTO_IPV4`, `NFPROTO_ARP`, and `NFPROTO_BRIDGE`. You will find only `NFPROTO_IPV4` in `net/ipv4/` in the kernel tree, and only `NFPROTO_IPV6` in `net/ipv6/`.

`priority` specifies where in the order of execution this hook gets executed. A few symbolic constants have been defined in `<linux/netfilter_ipv4.h>` and `<linux/netfilter_ipv6.h>` that can be used as a base for offsets, e.g. as the expression `NF_IP6_PRI_FILTER + 1` to get a hook that runs after the `filter` table has been processed. The value you pass in here is largely up to the module author, bound by the desired effect.

As with Xtables modules, the registration structure must not be `const` because fields in it will be modified by the implementation.

```

static struct nf_hook_ops myhook_ops __read_mostly = {
    .pf          = PF_INET6,
    .priority    = 1,
    .hooknum     = NF_INET_LOCAL_OUT,
    .hookfn      = myhook_fn,
};

static int __init myhook_init(void)
{
    return nf_register_hook(&myhook_ops);
}

static void __exit myhook_exit(void)
{
    nf_unregister_hook(&myhook_ops);
}

module_init(myhook_init);
module_exit(myhook_exit);

```

The hook infrastructure itself does not care about the `hooknum` either. A hook call specifies a certain hooknum and the hook module has to make something of it. It is generally used to encode the place the call originated; Xtables, NAT and connection tracking know of the five constants mentioned earlier (section 5.3), `NF_INET_{PRE,POST}_ROUTING`, `NF_INET_LOCAL_{IN,OUT}` and `NF_INET_LOCAL_OUT`.

10.3 Main function

```

static unsigned int myhook_fn(unsigned int hooknum, struct sk_buff *skb,
    const struct net_device *in, const struct net_device *out,
    int (*okfn)(struct sk_buff *))
{
    pr_info("Arr matey! - Captain Hook approves your packet!\n");
    return NF_ACCEPT;
}

```

The possible values a hook function can return are similar to those of a target. There is no `XT_RETURN`, because that one does not make sense here, there is no jump chain in which we could return. There is also no `XT_CONTINUE`, which, at best, would be equal to `NF_ACCEPT`, meaning that this hook allowed the packet to pass. Any other value indicates the packet has been “consumed”, as the `linux/netfilter.h` header file puts it. (So all of the `NF_*` constants as described in section 5.4 can be used.)

There are a few more not so common verdicts, mostly all internal to Netfilter, such as

- `NF_QUEUE` – used by `xt_NFQUEUE` to relay the packet to userspace for further processing
- `NF_STOLEN` – Whereas `NF_DROP` instructs Netfilter to drop the packet and free the `skb`, `NF_STOLEN` indicates that the hook has taken care of it so Netfilter only drops it. Netfilter forgetting about the packet does not imply the packet is lost — the hook may retransmit or delay it, for example.

- `NF_REPEAT` – causing a packet to be stopped and rerun through the current Netfilter hook. Connection tracking uses this to simplify its code path.
- `NF_STOP` – functionally the same as `NF_ACCEPT`. (Used as annotation?)

Part IV

Appendix

A Function reference

This list shall give a brief overview of the most common or useful functions you can use with Xtables modules. (We have even left out `const` and `unsigned` qualifiers to not bloat the prototypes too much.)

Byte swapping

- `htons`, `htonl`, `ntohs`, `ntohl`, `le16_to_cpu`, `le32_to_cpu`, `le64_to_cpu`, `be16_to_cpu`, `be32_to_cpu`, `be64_to_cpu`, `cpu_to_le16`, `cpu_to_le32`, `cpu_to_le64`, `cpu_to_be16`, `cpu_to_be32`, `cpu_to_be64`.

<linux/ip.h>

- `struct iphdr` – representation of the IPv4 header
- `ip_hdr(struct sk_buff *)` – returns a pointer to the IPv4 header

<linux/ipv6.h>

- `struct ipv6hdr` – representation of the IPv6 header
- `ipv6_hdr(struct sk_buff *)` – returns a pointer to the IPv6 header

<linux/kernel.h>

- `NIPQUAD(uint32_t)`, `NIPQUAD_FMT` – macros to be used when dumping IPv4 addresses with `printk`
- `NIP6(struct in_addr6)`, `NIP6_FMT` – macros to be used when dumping IPv6 addresses with `printk`

<linux/skbuff.h> The pointers in an `skb` are: `head`, `data`, `tail`, `end`. The allocated region spans `head`–`end` (not `tail`—odd naming here!), and the data region spans `data`–`end`.

- `skb_clone` – copy an `skb`, but the data remains shared (“hardlinked”)
- `skb_copy` – copy an `skb` and its data
- `skb_copy_expand` – copy an `skb` and its data, and additionally expand its size
- `skb_copy_bits` – scatter-gather bytes from a potentially non-linear `skb` and put them into a buffer
- `skb_header_pointer(struct sk_buff *skb, int offset, int length, void *buf)` – returns a pointer to the start of the layer-3 header. In case the `skb` is not linear, it will do a scatter-gather copy of the selected region into the provided buffer and returns `buf`. (`skb_header_pointer` can call `skb_copy_bits`.)

- `skb_linearize` – make an skb linear
- `skb_make_writable(struct sk_buff *, unsigned int length)` – make the skb writable for the given length; required for targets; returns NULL on failure. May change the pointers inside the skb.
- `skb_pull(struct sk_buff *, unsigned int length)` – “pull `skb->data` towards the right” — increments `skb->data` by `length` and decreases `skb->len` by the same amount.
- `skb_pull_tail(struct sk_buff *, unsigned int length)` – “pull `skb->tail` towards the right”
- `skb_push(struct sk_buff *, unsigned int length)` – “push `skb->data` towards the left” — decrements `skb->data` pointer by `length` and increases `skb->len` by the same amount.

Once again let me recommend (author?) [LinuxNetInt] which was very helpful in understanding the skb operations.

<linux/netfilter/x_tables.h>

- `xt_(un)register_match(es)(struct xt_match *)`, `xt_(un)register_target(s)(struct xt_target *)` – (un)register matches/targets with the Xtables framework; functions return negative on failure.

<net/ip.h>

- `ip_hdrlen(struct sk_buff *)` – size of the IPv4 header in this skb. The skb must contain IPv4 data.

<net/ipv6.h>

- `ipv6_addr_cmp(struct in6_addr *, struct in6_addr *)` – compare two IPv6 addresses for equality; returns 0 if they match.
- `ipv6_masked_addr_cmp(struct in6_addr *, struct in6_addr *)` – compare two IPv6 addresses with mask; returns 0 if they match.
- `int ipv6_skip_exthdr(struct sk_buff *, int start, uint8_t *proto)` – locate the start of the first non-extension header beginning at `start`. The protocol of the non-extension header that was found is stored in `*proto`, and the function returns the offset.

<net/netfilter/nf_conntrack.h>

- `nf_ct_get(struct sk_buff *)` – get conntrack entry for a packet
- `nf_ct_l3proto_try_module_get(int family)`, `nf_ct_l3proto_module_put(int)` – request and release connection tracking module for layer-3 protocol (used by matches)

B Tricks and traps

B.1 Use of double exclamation mark

The C programming language has three (as far as the problem described here is concerned) binary operations (`&`, `|`, `^`), but only two logical operations (`&&`, `||`). Additionally, `false` is represented by the integer value 0, and `true` is represented by the integer value 1, however, all non-zero values also evaluate to `true`. If the binary XOR is used as a substitute for logical XOR, we must make sure that both operands are in the logical/boolean domain (0 or 1), not in the numeral domain (0..INT_MAX), or unwanted side effects happen. Consider

```
if ((foo == bar) ^ (flags & 0x80))
```

Assuming `foo` does equal `bar` (i.e. is `true`) and `flags` does have 0x80 set (i.e. evaluates to `true`), the result of the binary XOR operation will be 0x81 (which also evaluates to `true`). Using a double negation “`!!`”, 0x80 is mapped into the logical domain (`!!0x80 = !0 = 1`), and so, `1 ^ 1` will yield 0, which evaluates to `false`.

```
if ((foo == bar) ^ !(flags & 0x80))
```

C Kernel/Xtables-addons differences

This is a list of compatibility decisions for Xtables-addons 1.27:

- Xtables-addons targets always use `struct sk_buff **` (double indirection) as their first argument irrespective of the real Xtables signature, which would have been `struct sk_buff *` from 2.6.24 onwards.
- 2.6.25 introduced `NF_INET_` constants. Xt-a provides them for older kernels.
- 2.6.29 obsoleted `NIPQUAD_FMT/NIP6_FMT`, `NIPQUAD/NIP6`; Xt-a continues to provide these, since the new printk specifiers `%pI4/%pI6` are not available on older kernels.

References

- [Chaostables] “Detecting and deceiving network scans”, Chaostables implementation document
Jan Engelhardt, 2006-12-31
<http://jengelh.medozas.de/documents/Chaostables.pdf>
- [ElseHarmful] “Else considered harmful”
2011-02-07, <http://iq0.com/notes/deep.nesting.html>
- [Git] Git (homepage)
Linus Torvalds, Junio C. Hamano, et al.
<http://git.or.cz/>
- [GitJB] “How to manage patches with Git and Quilt”
James Bottomley, FreedomHEC video from 2007-05-18
<http://linuxworld.com/video/>
- [LinuxNetInt] ”Understanding Linux Network Internals”, by Christian Benvenuti, distributed by O’Reilly
1st Edition, ISBN 0-596-00255-6, <http://oreilly.com/catalog/9780596002558/>

- [PacketFlow] “Netfilter packet flow; hook/table ordering”
Jan Engelhardt, graphic from 2008-02-07
<http://jengelh.medozas.de/images/nf-packet-flow.png>
- [QuiltAG] “Surviving with many patches, or, introduction to quilt”
2005-06-12, Andreas Grünbacher
<http://suse.de/~agruen/quilt.pdf>
- [QuiltFM] Quilt Freshmeat.net page
<http://freshmeat.net/p/quilt/>
- [QuotaOnSMP] “Re: Quota on SMP AGAIN”
Patrick McHardy, mail from 2007-12-30
<http://marc.info/?l=netfilter-devel&m=119903624211253&w=2>
- [ReadMostly] “Re: RFC: remove __read_mostly”
Eric Dumazet, mail from 2007-12-13
<http://lkml.org/lkml/2007/12/13/496>
- [RFC862] RFC 862: Echo Protocol
J. Postel, May 1983
<http://tools.ietf.org/html/rfc862>
- [StGit] Stacked Git, alternative to Quilt that directly integrates with Git
Catalin Marinas
<http://procode.org/stgit/>