Writing your own Xtables module

Nicolas Bouliane, Jan Engelhardt

February 21, 2008

Abstract

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 module. By writing your new extension, you can match, mangle, track and give faith to a given packet. In fact, you can do almost everything you want in this world. Beware that a little error in a kernel module can crash the computer.

We will explain the skeletal structure of an Xtables match module which will match a packet according to source and/or destination address. This way, we 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 have C programming skills.

Additionally, we will also cover Xtables targets a bit, as well as advanced topics and pitfalls that programmers should be aware of when writing a serious or complex module.

About the authors

Nicolas is a young warrior in the free software community. He is a GNU/Linux addict since the day he installed it on his computer in 1998. He spend his time studying the Linux networking stack, writing free softwares and attending at Linux-related conferences like OLS. When he is not in front of his computer, he likes watching sci-fi movies, playing chess and listening Richard Stallman's talk. He wrote the original "How to write your own Netfilter modules" in February 2005.

Jan uses Linux since fall 1999, and is involved in kernel hacking since 2003. His current focus is on Netfilter and iptables. This document has been extensively updated by him in 2008.

The authors can be reached at:

Nicolas Bouliane <acidfu (at) people.netfilter.org>, Jan Engelhardt <jengelh (at) computergmbh.de>

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.

If you have questions about writing add-on modules, or just have about any other Netfilter, Xtables or iptables development related question, you are welcome to join the netfilter-devel mailing list.

Jan Engelhardt, 2008-01-08

Contents

1 Prerequisites

A warning before we start: this document contains information about recently developed APIs. It focuses on the 2.6.25 Netfilter API, but also describes differences to the previous 2.6.23 so that you have some room. 2.6.25-rc1 is estimated to hit the shelf in mid-February 2008.

We believe that providing information about the newest development APIs makes this reference one that is recent for a while.

Jan presented a new package called xtables-addons at the end of January 2008 similar to the old patch-o-matic(-ng) (with the intention to replace p-o-m), where you can easily add new extensions without having to bother about the build infrastructure such as Makefiles. It also provides API compatibility glue code so that you can write extensions using the 2.6.25 API and get it running on older kernels. Currently, xtables-addons has glue code to provide backwards compatibility up to 2.6.19.

For iptables, we assume 1.4.0-svn7294 as a minimum, but xtables-1.5.1 (an improved iptables package) is highly recommended. `xtables-1.5.1.tar.bz2` may be found at `http://dev.computergmbh.de/files/xtables/`. You can also find xtables-addons at this location.

2 Nomenclature

By convention, names of Xtables matches are always lower-case, and names of Xtables targets always upper-case. Xtables modules are typically prefixed with xt_, forming something like xt_MARK for a target, and xt_connmark for a match.

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 and arp_tables are the kernel modules providing protocol-specific tables for the iptables, ip6tables and arptables tools.

ip, ip6 and arp table modules traditionally used distinct prefixes, according to their subsystem. They were ipt_, ip6t_ and arpt_, respectively. Use of these is fading and should probably be avoided for new modules. Nevertheless, the traditional names still need to be supported, because the iptables, ip6tables and arptables userspace utilities will automatically load kernel modules with this prefix rather than xt_ (let's hope for a change). To do so, just add one or more of:

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

to the source code of your module — ideally only those which you support. If you 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 named by their logical module name, i.e. `xt_mymatch.c` for the Xtables match (kernel) module; the only current exception is xt_TOS and xt_tos which are bundled into `xt_DSCP.c` and `xt_dscp.c`, respectively, because of their close nature. As long as it has the appropriate `MODULE_ALIAS`es, to ensure it can be loaded — if it is not already statically compiled into the kernel — there is no hard requirement on the filename.

Then there is also ebtables, which uses the ebt_ prefix for its modules, since it does not (yet) make use of Xtables modules. It also does not depend on the Xtables codebase. ebtables works on Data Link Layer (OSI layer 2), one below the usual layer Xtables deals with.

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.

3 Xtables match

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. The most omnipresent matches are of course match by source or destination address, which are done inside Xtables rather than a dedicated 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 (e.g. xt_hashlimit).

In this section, we will be writing a simple IP address match (even if Xtables does a better job at it). All that is needed is to tell Xtables whether or not it matched. A match may not modify many structures — as you can see from the function prototypes later on, a lot of variables are declared `const`. Modifying any of this data should only be done in targets, which are discussed later, but there are of course exceptions.

3.1 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` – structure containing all match data
- `ipaddr_mt` – the match function
- `ipaddr_mt_check` – function to check for validity of parameters in our struct
- `ipaddr_mt_destroy` – function 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_mtinfo`*N* – structure for revision *N*

3.2 Header file

The header file for our match is where you describe the binary interface between userspace and the kernel module. While the original author, Nicolas, put the subsection about the header file at the end, at least Jan usually begins 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?"

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 end up with:

```
#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,
};
```

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.

You should not use types which do not have a fixed width — `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 struct is not the same on both ends. Instead, use `uint16_t`, `uint32_t` and `uint64_t`, or their signed counterparts, `int16_t`, `int32_t` and `int64_t`, respectively. `char` is defined to be of size 1, so it is safe to use. For clarity, use `char` for characters and strings and `int8_t` (or `uint8_t`) for numbers.

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

```
struct xt_ipaddr_mtinfo {
        union nf_inet_addr src, dst;
        uint8_t flags;
};

#endif /* _LINUX_NETFILTER_XT_IPADDR_H */
```

`union nf_inet_addr` is a compound introduced for Linux 2.6.25 that can store either an IPv4 or IPv6 address. 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 `AF_INET6` 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`).

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 will do so in section **??**.

3.3 xt_match structure

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 interesting to us are left out with (...).

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

        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)(void *matchinfo);
        ...
        struct module *me;
};
```

We will be using `union nf_inet_addr`, which will be defined from 2.6.25(-rc1) onwards as follows:

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

The union provides storage for either an IPv4 or IPv6 address (in various types actually, but `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.

3.4 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 do special-layouting of symbols, which actually helps optimizing cachelining [1].

```
static struct xt_match ipaddr_mt4_reg __read_mostly = {
```

`name` is the name of the match that you define. `XT_FUNCTION_MAXNAMELEN` is currently `30`, so subtracting `1` and another one for the trailing `'\0'`, leaves 28 bytes 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 layer 3 protocol this `xt_match` structure supports. Using `0` or `AF_UNSPEC` for `family` will not act as a wildcard, as much as we would have liked. (Actually, the constant `NF_ARP`, which has value `0` in current kernels, is used for ARP, so it would also collide!) Both userspace and kernelspace must agree on the same (name, revision, family, size) 4-tuple for a `xt_match` to be successfully used.

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

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 not any 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, so xt_owner sets `hooks`. We will cover `hooks` in deeper detail in section 6.3 about Xtables targets.

`proto`, as far as matches are concerned, is primarily used by 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.

Only `hooks` is a bitmask; `table` and `proto` are single-value fields. 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 condition in the `checkentry` function (see section 3.6).

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 inside `ipaddr_mt_check` and `ipaddr_mt_destroy` in our sample module.

```
        .match      = ipaddr_mt,
        .checkentry = ipaddr_mt_check,
        .destroy    = ipaddr_mt_destroy,
        .me         = THIS_MODULE,
};
```

The last line containing `THIS_MODULE` is used for the Linux kernel module infrastructure; among other things, it servers 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)
{
        printk(KERN_INFO "xt_ipaddr: init!\n");
        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)
{
        printk(KERN_INFO "xt_ipaddr: exit!\n");
        xt_unregister_match(&ipaddr_mt4_reg);
}
```

`__init` and `__exit` are markers that cause the functions to be emit 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 macros:

```
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");
```

3.5 Point of decision – match function

The Linux networking stack is sprinkled with Netfilter hooks. Thus, when a packet walks in, the stack passes the packet to the appropriate hook which iterates through each table, which iterates through each rule, which (phew) 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,
    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)
{
```

`in` and `out` are the network devices through which the packet came in or went out; they may be `NULL` in certain hooks. `in` is available in PREROUTING, INPUT and FORWARD, while `out` is available in FORWARD, OUTPUT and POSTROUTING. `match` points to the structure through which the match function was invoked. It can be used, for example, to figure out whether the code was called with IPv4 or IPv6 as intention was called is used when you want to combine code (xt_connlimit for an example of this). `matchinfo` is the data block copied from userspace, and here we map it:

```
const struct xt_ipaddr_mtinfo *info = matchinfo;
```

`skb` contains the packet we want to look at. For more information about this powerful structure used everywhere in the Linux networking stack, see Harald Welte's article [2] about it. You need to include `<linux/skbuff.h>` for it.

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>`.

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

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)) {
                printk(KERN_NOTICE "src IP - no match\n");
                return false;
        }
```

For an explanation of the use of "!!", see the appendix.

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)) {
                        printk(KERN_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.

```
        return true;
}
```

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

```
op = skb_header_pointer(skb, protoff + sizeof(struct tcphdr),
                        optlen, _opt);
if (op == NULL) {
        *hotdrop = true;
        return false;
}
```

The protoff argument to our match function contains the layer-4 protocol offset in the skb, in this case the TCP header. Calling skb_header_pointer(skb, 0, ...) with zero as second argument will return the start of the layer-3 header.

3.6 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 iptables 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 such as xt_connlimit, xt_conntrack, and a few more. What's more, this function may be used to allocate some heap space — more on this in section 4.3. This function is called when you try to add a rule, but it happens before the rule is actually inserted.

```
static bool ipaddr_mt_check(const char *table, const void *entry,
    const struct xt_entry *match, void *matchinfo,
    unsigned int hook_mask)
{
        const struct xt_ipaddr_mtinfo *info = matchinfo;

        printk(KERN_INFO "xt_ipaddr: Added a rule with -m ipaddr in "
               "the %s table, hook 0x%x\n", table, hook_mask);
        return true;
}
```

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 6.5.

3.7 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.

```
static void ipaddr_mt_destroy(const struct xt_match *match,
    void *matchinfo)
{
        printk(KERN_INFO "One rule with ipaddr match got removed");
}
```

3.8 IPv6 support

IPv6 is slowly 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 net_device *in, const struct net_device *out,
    const struct xt_match *match, const void *matchinfo, int offset,
    unsigned int protoff, bool *hotdrop)
{
        const struct xt_ipaddr_mtinfo *info = 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    = AF_INET6,
        .match     = ipaddr_mt6,
        .matchsize = sizeof(struct xt_ipaddr_mtinfo),
        .me        = THIS_MODULE,
};
```

and call `xt_register_match(&ipaddr_mt6_reg)` next to the existing registration 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 matches 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 discussed later in section 4.1.

3.9 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.

3.9.1 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 `obj-m`, 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.

Kbuild:

```
obj-m += xt_ipaddr.o
```

mconfig:

```
build_ipaddr=m
```

3.9.2 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/sources` 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.).

3.9.3 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 Quilt [3, 4] or Git [5, 6] if you intend on submitting patches for your new module.

3.10 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. The advanced topic of per-match kernel-specific storage (also applies to targets) and the inherent SMP problems in the current model have been described.

4 Xtables tricks and traps

4.1 Registering multiple structures at once

As we have seen earlier in section 3.8, trying to register multiple structures at once can become a tedius 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    = AF_INET,
                .match     = ipaddr_mt4,
                .matchsize = sizeof(struct xt_ipaddr_mtinfo),
                .me        = THIS_MODULE,
        },
        {
                .name      = "ipaddr",
                .revision  = 0,
                .family    = AF_INET6,
                .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));
}
```

4.2 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 NAT use the `nf_nat_setup_info` symbol from `nf_conntrack.ko`. The dependencies between kernel modules are computed at compile time[1]. modprobe[2] 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 then calls modprobe itself). Run-time dependencies are allowed to fail to resolve, and code using such either tries something else or aborts gracefully.

```
static int conntrack_mt_check(const char *tablename, const void *entry,
    const struct xt_match *match, void *matchinfo, unsigned int hookmask)
{
        return nf_ct_l3proto_try_module_get(match->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-7 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.

`nf_ct_l3proto_module_try_get` increases the reference count of the layer-3 protocol module so that it cannot be removed using rmmod 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_match *match,
    void *matchinfo)
```

[1] Actually, link time — it is when modpost runs. But I chose the "compile-time" wording because modprobe is also a spot in time when linking happens.

[2] insmod does not do automatically load dependencies and thus often "fails" because symbols from dependent modules are not yet registered in the kernel.

```
{
        nf_ct_l3proto_module_put(match->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 (it is listed after all — has a symbol dependency) and nf_nat (it seems to irregularly grab nf_conntrack_ipv4 however).

```
Module              Used by
iptable_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
```

4.3 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:

```
struct xt_quota_mtinfo {
        uint32_t flags;

        /* Used internally by the kernel */
        uint64_t quota __attribute__((aligned(8)));
        struct xt_quota_mtinfo *master;
};
```

The first kernel-only variable should be aligned to the largest type, which is currently 64-bit (8 bytes). To do so, you use the `aligned` attribute — regardless of type, that is, `aligned(8)` must also be used for smaller types such as `uint8_t` or pointers.

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 hypothethical xt_bigipaddr match that records the timestamps of the eight most recent processed packets[3]:

```
struct xt_bigipaddr_state {
        union nf_inet_addr seen[8];
};

struct xt_bigipaddr_mtinfo {
        uint16_t match_flags, invert_flags;
```

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

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

static bool xt_bigipaddr_check(const char *table, const void *entry,
    const struct xt_match *match, void *matchinfo,
    unsigned int hook_mask)
{
        struct xt_bigipaddr_mtinfo *info = matchinfo;

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

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

The reason why xt_quota has the `quota` member in the `struct xt_quota_mtinfo` is because userspace examines this field. If the field was hidden behind a pointer, userspace could not access it, because kernel pointers are invalid in userspace — the kernel can only do so-called "shallow copies"[4]. An example for a hidden private structure is `struct xt_hashlimit_mtinfo1` and its `hinfo` member, which does not need to be exported to userspace in any way. Additionally, if the private data set is small, an extra kmalloc may not be worth the hassle, but this depends on a per-case basis.

4.4 SMP problems

You might have noticed the ominuous `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 [7]. This obviously creates a difficult decision: which `struct xt_quota_mtinfo` to update?

In the `check` function, `info->master` is set to `info`, i.e. to point to itself. Now when the struct is duplicated, the duplicates' addresses may change, but the `info->master` member in all duplicates still has the same value, and hence points to the original struct. Then it is easy to just update the master's counters:

```
struct xt_quota_mtinfo *q = matchinfo;
q = q->master;
q->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`'). Neither does this play well with

[4]Compare with "deep copies", where pointers are followed.

CPU hotplug — the CPU on which the master structure is may just go offline — both in theory and for sure on the big iron boxes. But these are problems that cannot be solved within the scope of this document.

5 The iptables extension module

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.

5.1 xtables_match structure

At first, some basic structures from `<xtables.h>`. We will see later in this text what the purpose of each field is.

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

        size_t size;
        size_t userspacesize;

        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;
        ...
};
```

5.2 Extension initialization

We initialize the common fields in the `xtables_match` structure.

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

`version` is always initialized to `IPTABLES_VERSION`. This is to avoid loading old modules in `/usr/libexec/iptables/modules` 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*, *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 layer-3 protocol this match operates on, in this case IPv4 (`AF_INET`), or IPv6 (`AF_INET6`). `family` cannot be left out, `0` (`AF_UNSPEC`) as a value will currently not make it act like a wildcard for all protocols.

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

`size` denotes the size of our private structure in total. `userspacesize` specifies the part of the structure that is relevant to userspace-kernelspace exchange and bitness conversion. Since you should avoid using types of variadic size (see section 3.2), bitness conversion will not be of an issue. Usually, both `size` and `userspacesize` are the same, but there are exceptions such as xt_rateest where the kernel module keeps additional information for itself.

```
        .size           = XT_ALIGN(sizeof(struct xt_ipaddr_mtinfo)),
        .userspacesize = 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_m` 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);
void _init(void)
{
        xtables_register_match(&ipaddr_mt_reg);
}
```

When iptables is built, this will expand to:

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

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

Note the extra prototype right above the function's body. It is needed to shut up the compiler because it is not provided in any header file. There is not much that can go wrong, so it should be ok to do it this way.

5.3 Dumping rules – save function

If we have a rule set 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 ",
                       ipaddr_to_numeric(&info->ipaddr_src));
        }
```

Note that `ipaddr_to_numeric` uses a static buffer, so you may not call it more than once in a statement. 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 ", ipaddr_to_numeric(&info->dst);
        }
}
```

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

5.4 Status display – print function

In the same philosophy as the previous one, this function aims to print information about the rule. It is called by '`iptables -L`', and you are free to output whatever you want, and how you want, but it should be human-readable of course.

```
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 ", ipaddr_to_numeric(&info->src));
                else
                        printf("%s ", ipaddr_to_anyname(&info->src));
        }

        if (info->flags & XT_IPADDR_DST) {
                printf("dst IP ");
                if (info->flags & XT_IPADDR_DST_INV)
                        printf("! ");
                if (numeric)
                        printf("%s ", ipaddr_to_numeric(&info->dst));
                else
                        printf("%s ", ipaddr_to_anyname(&info->dst));
        }
}
```

Here, we use `addr_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.

5.5 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 not it?).

```
struct xt_ipaddr_mtinfo *info = (void *)(*match)->data;
struct in_addr *addrp;
unsigned int addrs;
```

Each argument corresponds to a single value, so we can do specific actions according to the inputed 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 `exit_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)
                exit_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 `ipparse_hostnetwork`, which can take either a hostname or IP address, and will return the result in a newly allocated buffer, as well as the number of IP addresses that the DNS lookup resolved (in `addrs`). We keep it simple here and omit the check for `addrs > 1`, even if our libxt_ipaddr module can only deal with one address at a time; we just pick the first address.

```
        if (invert)
                info->flags |= XT_IPADDR_SRC_INV;
        addrp = ipparse_hostnetwork(optarg, &addrs);
        memcpy(&info->src.in, addrp, sizeof(info->src.in));
        free(addrp);
        return true;
```

For demonstrational purposes, we will use `numeric_to_ipaddr` instead for the destination address. It transforms exactly one IPv4 address (no hostname!) into a 32-bit entity:

```
case '2': /* --ipdst */
        if (*flags & XT_IPADDR_DST)
                exit_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;
                numeric_to_ipaddr(optarg, &info->dst.in);
                return true;
        }
        return false;
}
```

Everytime an option was recognized, the parse function should return `true`. 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, everytime 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. This is why a command like '`iptables -p tcp -m multiport --dport 22,80`' will (correctly) call libxt_multiport's `--dport` handler (actually `--dports`; abbreviations are handled by `getopt`) instead the one of libxt_tcp.

5.6 Option validation – check function

This function is a kind of last chance for sanity check. It is called when the user enters a new rule, right 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)
                exit_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 these combos.

5.7 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'},
        {},
};
```

5.8 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 we did above).

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

        inet_pton(AF_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. The initialization is often not needed because the memory pointed to by `match->data` is already zeroed so our iptables extension does not need to take care of clearing `info->flags` before being able to use it in the parse function, for example.

5.9 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 5.11).

```
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"
);
}
```

5.10 IPv6 support

As with the kernel module, you will also want to add IPv6 support in the iptables extension. For that, we need a separate `struct xtables_match`. (There is currently no plural function like `xt_register_matches` from the kernel.)

```
static struct xtables_match ipaddr_mt6_reg = {
        .version       = IPTABLES_VERSION,
        .name          = "ipaddr",
        .revision      = 0,
        .family        = AF_INET6,
        .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)
{
        switch (c) {
        case '1': /* --ipsrc */
                if (*flags & XT_IPADDR_SRC)
                        exit_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;
                numeric_to_ip6addr(optarg, &info->src.in6);
                return true;
        }
        return false;
}
```

I have left out the `case '2'`, you can surely add it yourself. The only interesting change here is that we use `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 `ip6addr_to_numeric` and `ip6addr_to_anyname`, respectively. Add registering the `ipaddr_mt6_reg` structure to `_init`, and you should be done:

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

5.11 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--ipsrc\fR \fIaddr\fR
Match packets that have \fIaddr\fR as source address.
.TP
\fB--ipdst\fR \fIaddr\fR
Match packets that have \fIaddr\fR as destination address.
.PP
The ipaddr module serves only as a demonstration. It is equivalent to
the iptables \fB-s\fR and \fB-d\fR 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 \fR for normal. .TP will do an indentation appropriate for option and description, and .PP will return to the default paragraph indentation.

5.12 Building the extension

5.12.1 Using the Xtables-addons package

Place libxt_ipaddr.c into the extensions/ directory, and add a line to the mconfig file (if you have not done so yet):

```
build_ipaddr=m
```

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 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 this directory. (Note that this will currently skip looking for modules in the default extension directory, so you might not have access to the usual extensions!)

5.12.2 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 as simple as:

```
CFLAGS = -O2 -Wall
libxt_ipaddr.so: libxt_ipaddr.o
        gcc -shared -o $@ $^;
```

5.12.3 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 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` variable:

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

then you can test ipaddr:

```
./iptables -m ipaddr -h
./iptables -A INPUT -m ipaddr --ipsrc 192.0.2.137
```

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.

5.13 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.

6 Xtables target

Targets can be really be versatile. Common categories and examples are, in some sort of descending order:

- mangling the packet payload – xt_TCPOPTSTRIP, xt_TCPMSS, ipt_XOR
- setting up NAT mappings – ipt_MASQUERADE, ipt_NETMAP
- triggering a hard action – ipt_SYSRQ
- replying to packets – ipt_REJECT, xt_DELUDE
- changing skb, packet or connection parameters (i.e. something that is not visible on the wire) – xt_CLASSIFY, xt_CONNMARK

- changing parameters unrelated to the packet stream (e.g. sysctl) – *yet to be seen*
- tracking the packet, e.g. for statistical purposes – xt_RATEEST (most often, matches are used instead for simplicity)
- just watching packets – ip6t_LOG

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

6.1 Naming convention

Just like for matches (see section 3.1), 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. We then have the default naming, here with xt_CLASSIFY:

- `classify_tg_reg` – structure containing all match data
- `classify_tg` (or `classify_tg4`, `classify_tg6` if it used distinct functions) – the match function
- `classify_tg_check` – function to check for validity of parameters in our struct
- `classify_tg_destroy` – function when rule is deleted
- `struct xt_classify_tginfo` and `struct xt_classify_tginfo`*N* – structure for our own data (for revision *N*)

I shall note here that the xt_CLASSIFY and other source code does not completely use these because of compatibility constraints (we have to keep old struct names for iptables).

6.2 xt_target structure

This is the `xt_target` structure and the fields you should care about. It is also defined in `<linux/netfilter/x_tables.h>`.

```
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 targetsize;
        unsigned int (*target)(struct sk_buff *skb,
                               const struct net_device *in,
                               const struct net_device *out,
                               unsigned int hooknum,
                               const struct xt_target *target,
```

```
                                const void *targinfo);
        bool (*checkentry)(const char *table, const void *entry,
                           const struct xt_target *target,
                           void *targinfo, unsigned int hook_mask);
        void (*destroy)(const struct xt_target *target, void *targinfo);
        ...
        struct module *me;
};
```

6.3 Module initialization

The structure looks quite the same as matches do (see section 3.4), so the initialization is straightforward. Let's have a look at xt_CLASSIFY:

```
static struct xt_target classify_tg_reg __read_mostly = {
        .name       = "CLASSIFY",
        .revision   = 0,
        .family     = AF_INET,
```

Here we actually see use of the `table` and `hooks` fields. Targets that modify the packet are usually limited to the mangle table, but there is no hard technical restriction requiring this. (xt_TCPMSS is such a case for example, more on it below.)

```
        .table      = "mangle",
```

`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 continues to use different flags (`NF_ARP_IN`, `NF_ARP_OUT`, `NF_ARP_FORWARD`, etc.). If `hooks` is not set, it is initialized to `0` by default, which implies that this target can be used in all chains.

```
        .hooks      = (1 << NF_INET_FORWARD) |
                      (1 << NF_INET_LOCAL_OUT) |
                      (1 << NF_INET_POST_ROUTING),
        .target     = classify_tg,
        .targetsize = sizeof(struct xt_classify_tginfo),
        .me         = THIS_MODULE,
};
```

As usual, we need to (un)register the target on module insertion/removal:

```
static int __init classify_tg_init(void)
{
        return xtables_register_match(&classify_tg_reg);
}

static void __exit classify_tg_exit(void)
{
        xtables_unregister_target(&classify_tg_reg);
}
```

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_unregiste`

6.4 Time for action – target function

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. Prior to 2.6.25, the target function uses an indirect skb pointer, `const struct sk_buff **pskb`; for those kernels, use `*pskb` when accessing the skb[5].

```
static unsigned int classify_tg(struct sk_buff *skb,
    const struct net_device *in, const struct net_device *out,
    unsigned int hooknum, const struct xt_target *target,
    const void *targinfo)
{
        const struct xt_classify_tginfo *info = targinfo;

        skb->priority = info->priority;
        return XT_CONTINUE;
}
```

Possible return values are:

- `XT_CONTINUE` – continue with next rule
- `NF_DROP` – stop traversal in the current table hook and drop packet
- `NF_ACCEPT` – stop traversal in the current table hook and accept packet

6.5 Rule validation – checkentry function

There is not much to say here. 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 3.6.

xt_TCPMSS 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 of setting the MSS from the FORWARD, OUTPUT and POSTROUTING chains, when an output route has been decided.

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

```
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 false;
```

6.6 Rule destruction – destroy function

Like with matches, targets can have a `destroy` function as a counterpart to `checkentry`.

6.7 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-protocol (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` 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.

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.)

<linux/ip.h>

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

<linux/ipv6.h>

- `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>

- `skb_copy_bits` – copy bytes from a skb to a buffer (needed because the skb might not be linear, though it seems that the layer-3 header is always available)
- `skb_header_pointer`(**struct sk_buff ***) – returns a pointer to the start of the layer-3 header.
- `skb_make_writable`(**struct sk_buff ***) – make the skb writable; required for targets; returns `NULL` on failure.
- `skb_tailroom` –

<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/ipv6.h>

- `ipv6_addr_cmp`(**struct in6_addr ***, **struct in6_addr ***) – compare two IPv6 addresses for equailty; 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.
- `ipv6_skip_exthdr` –

<net/netfilter/nf_conntrack.h>

- `nf_ct_get`(**struct sk_buff ***) – get conntrack entry for a packet
- `nf_ct_l3proto_try_module_get`/`put`(**int** *family*) – load connection tracking module for layer-3 protocol (used by matches)

B 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))
```

References

[1] "Re: RFC: remove `__read_mostly`"
Eric Dumazet, mail from 2007-12-13
`http://lkml.org/lkml/2007/12/13/496`

[2] Harald Welte, "skb — Linux network buffers"
Harald Welte, document from 2000-10-14
`ftp://ftp.gnumonks.org/pub/doc/skb-doc.html`

[3] Quilt Freshmeat.net page
`http://freshmeat.net/p/quilt/`

[4] "Surviving with many patches, or, introduction to quilt"
2005-06-12, Andreas Grünbacher
`http://suse.de/~agruen/quilt.pdf`

[5] Git homepage, `http://git.or.cz/`

[6] "How to manage patches with Git and Quilt"
James Bottomley, FreedomHEC video from 2007-05-18
`http://linuxworld.com/video/`

[7] "Re: Quota on SMP AGAIN"
Patrick McHardy, mail from 2007-12-30
`http://marc.info/?l=netfilter-devel&m=119903624211253&w=2`

[8] "Netfilter packet flow; hook/table ordering"
Jan Engelhardt, graphic from 2008-02-07
`http://jengelh.hopto.org/images/nf-packet-flow.png`