Next Previous Contents

4. Information for Programmers

I'll let you in on a secret: my pet hamster did all the coding. I was just a channel, a `front' if you will, in my pet's grand plan. So, don't blame me if there are bugs. Blame the cute, furry one.

4.1 Understanding ip_tables

iptables simply provides a named array of rules in memory (hence the name `iptables'), and such information as where packets from each hook should begin traversal. After a table is registered, userspace can read and replace its contents using getsockopt() and setsockopt().

iptables does not register with any netfilter hooks: it relies on other modules to do that and feed it the packets as appropriate; a module must register the netfilter hooks and ip_tables separately, and provide the mechanism to call ip_tables when the hook is reached.

ip_tables Data Structures

For convenience, the same data structure is used to represent a rule by userspace and within the kernel, although a few fields are only used inside the kernel.

Each rule consists of the following parts:

1. A `struct ipt_entry'.
2. Zero or more `struct ipt_entry_match' structures, each with a variable amount (0 or more bytes) of data appended to it.
3. A `struct ipt_entry_target' structure, with a variable amount (0 or more bytes) of data appended to it.

The variable nature of the rule gives a huge amount of flexibility for extensions, as we'll see, especially as each match or target can carry an arbitrary amount of data. This does create a few traps, however: we have to watch out for alignment. We do this by ensuring that the `ipt_entry', `ipt_entry_match' and `ipt_entry_target' structures are conveniently sized, and that all data is rounded up to the maximal alignment of the machine using the IPT_ALIGN() macro.

The `struct ipt_entry' has the following fields:

1. A `struct ipt_ip' part, containing the specifications for the IP header that it is to match.
2. An `nf_cache' bitfield showing what parts of the packet this rule examined.
3. A `target_offset' field indicating the offset from the beginning of this rule where the ipt_entry_target structure begins. This should always be aligned correctly (with the IPT_ALIGN macro).
4. A `next_offset' field indicating the total size of this rule, including the matches and target. This should also be aligned correctly using the IPT_ALIGN macro.
5. A `comefrom' field used by the kernel to track packet traversal.
6. A `struct ipt_counters' field containing the packet and byte counters for packets which matched this rule.

The `struct ipt_entry_match' and `struct ipt_entry_target' are very similar, in that they contain a total (IPT_ALIGN'ed) length field (`match_size' and `target_size' respectively) and a union holding the name of the match or target (for userspace), and a pointer (for the kernel).

Because of the tricky nature of the rule data structure, some helper routines are provided:

ipt_get_target()

This inline function returns a pointer to the target of a rule.

IPT_MATCH_ITERATE()

This macro calls the given function for every match in the given rule. The function's first argument is the `struct ipt_match_entry', and other arguments (if any) are those supplied to the IPT_MATCH_ITERATE() macro. The function must return either zero for the iteration to continue, or a non-zero value to stop.

IPT_ENTRY_ITERATE()

This function takes a pointer to an entry, the total size of the table of entries, and a function to call. The functions first argument is the `struct ipt_entry', and other arguments (if any) are those supplied to the IPT_ENTRY_ITERATE() macro. The function must return either zero for the iteration to continue, or a non-zero value to stop.

ip_tables From Userspace

Userspace has four operations: it can read the current table, read the info (hook positions and size of table), replace the table (and grab the old counters), and add in new counters.

This allows any atomic operation to be simulated by userspace: this is done by the libiptc library, which provides convenience "add/delete/replace" semantics for programs.

Because these tables are transferred into kernel space, alignment becomes an issue for machines which have different userspace and kernelspace type rules (eg. Sparc64 with 32-bit userland). These cases are handled by overriding the definition of IPT_ALIGN for these platforms in `libiptc.h'.

ip_tables Use And Traversal

The kernel starts traversing at the location indicated by the particular hook. That rule is examined, if the `struct ipt_ip' elements match, each `struct ipt_entry_match' is checked in turn (the match function associated with that match is called). If the match function returns 0, iteration stops on that rule. If it sets the `hotdrop' parameter to 1, the packet will also be immediately dropped (this is used for some suspicious packets, such as in the tcp match function).

If the iteration continues to the end, the counters are incremented, the `struct ipt_entry_target' is examined: if it's a standard target, the `verdict' field is read (negative means a packet verdict, positive means an offset to jump to). If the answer is positive and the offset is not that of the next rule, the `back' variable is set, and the previous `back' value is placed in that rule's `comefrom' field.

For non-standard targets, the target function is called: it returns a verdict (non-standard targets can't jump, as this would break the static loop-detection code). The verdict can be IPT_CONTINUE, to continue on to the next rule.

4.2 Extending iptables

Because I'm lazy, `iptables` is fairly extensible. This is basically a scam to palm off work onto other people, which is what Open Source is all about (cf. Free Software, which as RMS would say, is about freedom, and I was sitting in one of his talks when I wrote this).

Extending `iptables` potentially involves two parts: extending the kernel, by writing a new module, and possibly extending the userspace program `iptables`, by writing a new shared library.

The Kernel

Writing a kernel module itself is fairly simple, as you can see from the examples. One thing to be aware of is that your code must be re-entrant: there can be one packet coming in from userspace, while another arrives on an interrupt. In fact in SMP there can be one packet on an interrupt per CPU in 2.3.4 and above.

The functions you need to know about are:

init_module()

This is the entry-point of the module. It returns a negative error number, or 0 if it successfully registers itself with netfilter.

cleanup_module()

This is the exit point of the module; it should unregister itself with netfilter.

ipt_register_match()

This is used to register a new match type. You hand it a `struct ipt_match', which is usually declared as a static (file-scope) variable.

ipt_register_target()

This is used to register a new type. You hand it a `struct ipt_target', which is usually declared as a static (file-scope) variable.

ipt_unregister_target()

Used to unregister your target.

ipt_unregister_match()

Used to unregister your match.

One warning about doing tricky things (such as providing counters) in the extra space in your new match or target. On SMP machines, the entire table is duplicated using memcpy for each CPU: if you really want to keep central information, you should see the method used in the `limit' match.

New Match Functions

New match functions are usually written as a standalone module. It's possible to have these modules extensible in turn, although it's usually not necessary. One way would be to use the netfilter framework's `nf_register_sockopt' function to allows users to talk to your module directly. Another way would be to export symbols for other modules to register themselves, the same way netfilter and ip_tables do.

The core of your new match function is the struct ipt_match which it passes to `ipt_register_match()'. This structure has the following fields:

list

This field is set to any junk, say `{ NULL, NULL }'.

name

This field is the name of the match function, as referred to by userspace. The name should match the name of the module (i.e., if the name is "mac", the module must be "ipt_mac.o") for auto-loading to work.

match

This field is a pointer to a match function, which takes the skb, the in and out device pointers (one of which may be NULL, depending on the hook), a pointer to the match data in the rule that is worked on (the structure that was prepared in userspace), the IP offset (non-zero means a non-head fragment), a pointer to the protocol header (i.e., just past the IP header), the length of the data (ie. the packet length minus the IP header length) and finally a pointer to a `hotdrop' variable. It should return non-zero if the

packet matches, and can set `hotdrop' to 1 if it returns 0, to indicate that the packet must be dropped immediately.

checkentry

This field is a pointer to a function which checks the specifications for a rule; if this returns 0, then the rule will not be accepted from the user. For example, the "tcp" match type will only accept tcp packets, and so if the `struct ipt_ip' part of the rule does not specify that the protocol must be tcp, a zero is returned. The tablename argument allows your match to control what tables it can be used in, and the `hook_mask' is a bitmask of hooks this rule may be called from: if your match does not make sense from some netfilter hooks, you can avoid that here.

destroy

This field is a pointer to a function which is called when an entry using this match is deleted. This allows you to dynamically allocate resources in checkentry and clean them up here.

me

This field is set to `THIS_MODULE', which gives a pointer to your module. It causes the usage-count to go up and down as rules of that type are created and destroyed. This prevents a user removing the module (and hence cleanup_module() being called) if a rule refers to it.

New Targets

If your target alters the packet (ie. the headers or the body), it must call skb_unshare() to copy the packet in case it is cloned: otherwise any raw sockets which have a clone of the skbuff will see the alterations (ie. people will see wierd stuff happening in tcpdump).

New targets are also usually written as a standalone module. The discussions under the above section on `New Match Functions' apply equally here.

The core of your new target is the struct ipt_target that it passes to ipt_register_target(). This structure has the following fields:

list

This field is set to any junk, say `{ NULL, NULL }'.

name

This field is the name of the target function, as referred to by userspace. The name should match the name of the module (i.e., if the name is "REJECT", the module must be "ipt_REJECT.o") for auto-loading to work.

target

This is a pointer to the target function, which takes the skbuff, the hook number, the input and output device pointers (either of which may be NULL), a pointer to the target data, and the position of the rule in the table. The target function may return either IPT_CONTINUE (-1) if traversing should continue, or a netfilter verdict (NF_DROP, NF_ACCEPT, NF_STOLEN etc.).

checkentry

This field is a pointer to a function which checks the specifications for a rule; if this returns 0, then the rule will not be accepted from the user.

destroy

This field is a pointer to a function which is called when an entry using this target is deleted. This allows you to dynamically allocate resources in checkentry and clean them up here.

me

This field is set to `THIS_MODULE', which gives a pointer to your module. It causes the usage-count to go up and down as rules with this as a target are created and destroyed. This prevents a user removing the module (and hence cleanup_module() being called) if a rule refers to it.

New Tables

You can create a new table for your specific purpose if you wish. To do this, you call `ipt_register_table()', with a `struct ipt_table', which has the following fields:

list

This field is set to any junk, say `{ NULL, NULL }'.

name

This field is the name of the table function, as referred to by userspace. The name should match the name of the module (i.e., if the name is "nat", the module must be "iptable_nat.o") for auto-loading to work.

table

This is a fully-populated `struct ipt_replace', as used by userspace to replace a table. The `counters' pointer should be set to NULL. This data structure can be declared `__initdata' so it is discarded after boot.

valid_hooks

This is a bitmask of the IPv4 netfilter hooks you will enter the table with: this is used to check that those entry points are valid, and to calculate the possible hooks for ipt_match and ipt_target `checkentry()' functions.

lock

This is the read-write spinlock for the entire table; initialize it to RW_LOCK_UNLOCKED.

private

This is used internally by the ip_tables code.

Userspace Tool

Now you've written your nice shiny kernel module, you may want to control the options on it from userspace. Rather than have a branched version of `iptables` for each extension, I use the very latest 90's technology: furbies. Sorry, I mean shared libraries.

New tables generally don't require any extension to `iptables`: the user just uses the `-t' option to make it use the new table.

The shared library should have an `_init()' function, which will automatically be called upon loading: the moral equivalent of the kernel module's `init_module()' function. This should call `register_match()' or `register_target()', depending on whether your shared library provides a new match or a new target.

You need to provide a shared library: this can be used to initialize part of the structure, or provide additional options. I now insist on a shared library even if it doesn't do anything, to reduce problem reports where the

shares libraries are missing.

There are useful functions described in the `iptables.h' header, especially:

check_inverse()

checks if an argument is actually a `!', and if so, sets the `invert' flag if not already set. If it returns true, you should increment optind, as done in the examples.

string_to_number()

converts a string into a number in the given range, returning -1 if it is malformed or out of range. `string_to_number' rely on `strtol' (see the manpage), meaning that a leading "0x" would make the number be in Hexadecimal base, a leading "0" would make it be in Octal base.

exit_error()

should be called if an error is found. Usually the first argument is `PARAMETER_PROBLEM', meaning the user didn't use the command line correctly.

New Match Functions

Your shared library's _init() function hands `register_match()' a pointer to a static `struct iptables_match', which has the following fields:

next

This pointer is used to make a linked list of matches (such as used for listing rules). It should be set to NULL initially.

name

The name of the match function. This should match the library name (eg "tcp" for `libipt_tcp.so').

version

Usually set to the IPTABLES_VERSION macro: this is used to ensure that the `iptables` binary doesn't pick up the wrong shared libraries by mistake.

size

The size of the match data for this match; you should use the IPT_ALIGN() macro to ensure it is correctly aligned.

userspacesize

For some matches, the kernel changes some fields internally (the `limit' target is a case of this). This means that a simple `memcmp()' is insufficient to compare two rules (required for delete-matching-rule functionality). If this is the case, place all the fields which do not change at the start of the structure, and put the size of the unchanging fields here. Usually, however, this will be identical to the `size' field.

help

A function which prints out the option synopsis.

init

This can be used to initialize the extra space (if any) in the ipt_entry_match structure, and set any nfcache bits; if you are examining something not expressible using the contents of

`linux/include/netfilter_ipv4.h', then simply OR in the NFC_UNKNOWN bit. It will be called before `parse()'.

parse

This is called when an unrecognized option is seen on the command line: it should return non-zero if the option was indeed for your library. `invert' is true if a `!' has already been seen. The `flags' pointer is for the exclusive use of your match library, and is usually used to store a bitmask of options which have been specified. Make sure you adjust the nfcache field. You may extend the size of the `ipt_entry_match' structure by reallocating if necessary, but then you must ensure that the size is passed through the IPT_ALIGN macro.

final_check

This is called after the command line has been parsed, and is handed the `flags' integer reserved for your library. This gives you a chance to check that any compulsory options have been specified, for example: call `exit_error()' if this is the case.

print

This is used by the chain listing code to print (to standard output) the extra match information (if any) for a rule. The numeric flag is set if the user specified the `-n' flag.

save

This is the reverse of parse: it is used by `iptables-save' to reproduce the options which created the rule.

extra_opts

This is a NULL-terminated list of extra options which your library offers. This is merged with the current options and handed to getopt_long; see the man page for details. The return code for getopt_long becomes the first argument (`c') to your `parse()' function.

There are extra elements at the end of this structure for use internally by `iptables`: you don't need to set them.

New Targets

Your shared library's _init() function hands `register_target()' it a pointer to a static `struct iptables_target', which has similar fields to the iptables_match structure detailed above.

Using `libiptc'

`libiptc` is the iptables control library, designed for listing and manipulating rules in the iptables kernel module. While its current use is for the iptables program, it makes writing other tools fairly easy. You need to be root to use these functions.

The kernel tables themselves are simply a table of rules, and a set of numbers representing entry points. Chain names ("INPUT", etc) are provided as an abstraction by the library. User defined chains are labelled by inserting an error node before the head of the user-defined chain, which contains the chain name in the extra data section of the target (the builtin chain positions are defined by the three table entry points).

The following standard targets are supported: ACCEPT, DROP, QUEUE (which are translated to NF_ACCEPT, NF_DROP, and NF_QUEUE, respectively), RETURN (which is translated to a special IPT_RETURN value handled by ip_tables), and JUMP (which is translated from the chain name to an actual offset within the table).

When `iptc_init()' is called, the table, including the counters, is read. This table is manipulated by the `iptc_insert_entry()', `iptc_replace_entry()', `iptc_append_entry()', `iptc_delete_entry()', `iptc_delete_num_entry()', `iptc_flush_entries()', `iptc_zero_entries()', `iptc_create_chain()' `iptc_delete_chain()', and `iptc_set_policy()' functions.

The table changes are not written back until the `iptc_commit()' function is called. This means it is possible for two library users operating on the same chain to race each other; locking would be required to prevent this, and it is not currently done.

There is no race with counters, however; counters are added back in to the kernel in such a way that counter increments between the reading and writing of the table still show up in the new table.

There are various helper functions:

iptc_first_chain()

This function returns the first chain name in the table.

iptc_next_chain()

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

iptc_builtin()

Returns true if the given chain name is the name of a builtin chain.

iptc_first_rule()

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

iptc_next_rule()

This returns a pointer to the next rule in the chain: NULL means the end of the chain.

iptc_get_target()

This gets the target of the given rule. If it's an extended target, the name of that target is returned. If it's a jump to another chain, the name of that chain is returned. If it's a verdict (eg. DROP), that name is returned. If it has no target (an accounting-style rule), then the empty string is returned.

Note that this function should be used instead of using the value of the `verdict' field of the ipt_entry structure directly, as it offers the above further interpretations of the standard verdict.

iptc_get_policy()

This gets the policy of a builtin chain, and fills in the `counters' argument with the hit statistics on that policy.

iptc_strerror()

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

4.3 Understanding NAT

Welcome to Network Address Translation in the kernel. Note that the infrastructure offered is designed more for completeness than raw efficiency, and that future tweaks may increase the efficiency markedly. For the moment I'm happy that it works at all.

NAT is separated into connection tracking (which doesn't manipulate packets at all), and the NAT code itself. Connection tracking is also designed to be used by an iptables modules, so it makes subtle distinctions in states which NAT doesn't care about.

Connection Tracking

Connection tracking hooks into high-priority NF_IP_LOCAL_OUT and NF_IP_PRE_ROUTING hooks, in order to see packets before they enter the system.

The nfct field in the skb is a pointer to inside the struct ip_conntrack, at one of the infos[] array. Hence we can tell the state of the skb by which element in this array it is pointing to: this pointer encodes both the state structure and the relationship of this skb to that state.

The best way to extract the `nfct' field is to call `ip_conntrack_get()', which returns NULL if it's not set, or the connection pointer, and fills in ctinfo which describes the relationship of the packet to that connection. This enumerated type has several values:

IP_CT_ESTABLISHED

The packet is part of an established connection, in the original direction.

IP_CT_RELATED

The packet is related to the connection, and is passing in the original direction.

IP_CT_NEW

The packet is trying to create a new connection (obviously, it is in the original direction).

IP_CT_ESTABLISHED + IP_CT_IS_REPLY

The packet is part of an established connection, in the reply direction.

IP_CT_RELATED + IP_CT_IS_REPLY

The packet is related to the connection, and is passing in the reply direction.

Hence a reply packet can be identified by testing for >= IP_CT_IS_REPLY.

4.4 Extending Connection Tracking/NAT

These frameworks are designed to accommodate any number of protocols and different mapping types. Some of these mapping types might be quite specific, such as a load-balancing/fail-over mapping type.

Internally, connection tracking converts a packet to a "tuple", representing the interesting parts of the packet, before searching for bindings or rules which match it. This tuple has a manipulatable part, and a non-manipulatable part; called "src" and "dst", as this is the view for the first packet in the Source NAT world (it'd be a reply packet in the Destination NAT world). The tuple for every packet in the same packet stream in that direction is the same.

For example, a TCP packet's tuple contains the manipulatable part: source IP and source port, the non-manipulatable part: destination IP and the destination port. The manipulatable and non-manipulatable parts do not need to be the same type though; for example, an ICMP packet's tuple contains the manipulatable part: source IP and the ICMP id, and the non-manipulatable part: the destination IP and the ICMP type and code.

Every tuple has an inverse, which is the tuple of the reply packets in the stream. For example, the inverse of an ICMP ping packet, icmp id 12345, from 192.168.1.1 to 1.2.3.4, is a ping-reply packet, icmp id 12345,

from 1.2.3.4 to 192.168.1.1.

These tuples, represented by the `struct ip_conntrack_tuple', are used widely. In fact, together with the hook the packet came in on (which has an effect on the type of manipulation expected), and the device involved, this is the complete information on the packet.

Most tuples are contained within a `struct ip_conntrack_tuple_hash', which adds a doubly linked list entry, and a pointer to the connection that the tuple belongs to.

A connection is represented by the `struct ip_conntrack': it has two `struct ip_conntrack_tuple_hash' fields: one referring to the direction of the original packet (tuplehash[IP_CT_DIR_ORIGINAL]), and one referring to packets in the reply direction (tuplehash[IP_CT_DIR_REPLY]).

Anyway, the first thing the NAT code does is to see if the connection tracking code managed to extract a tuple and find an existing connection, by looking at the skbuff's nfct field; this tells us if it's an attempt on a new connection, or if not, which direction it is in; in the latter case, then the manipulations determined previously for that connection are done.

If it was the start of a new connection, we look for a rule for that tuple, using the standard iptables traversal mechanism, on the `nat' table. If a rule matches, it is used to initialize the manipulations for both that direction and the reply; the connection-tracking code is told that the reply it should expect has changed. Then, it's manipulated as above.

If there is no rule, a `null' binding is created: this usually does not map the packet, but exists to ensure we don't map another stream over an existing one. Sometimes, the null binding cannot be created, because we have already mapped an existing stream over it, in which case the per-protocol manipulation may try to remap it, even though it's nominally a `null' binding.

Standard NAT Targets

NAT targets are like any other iptables target extensions, except they insist on being used only in the `nat' table. Both the SNAT and DNAT targets take a `struct ip_nat_multi_range' as their extra data; this is used to specify the range of addresses a mapping is allowed to bind into. A range element, `struct ip_nat_range' consists of an inclusive minimum and maximum IP address, and an inclusive maximum and minimum protocol-specific value (eg. TCP ports). There is also room for flags, which say whether the IP address can be mapped (sometimes we only want to map the protocol-specific part of a tuple, not the IP), and another to say that the protocol-specific part of the range is valid.

A multi-range is an array of these `struct ip_nat_range' elements; this means that a range could be "1.1.1.1-1.1.1.2 ports 50-55 AND 1.1.1.3 port 80". Each range element adds to the range (a union, for those who like set theory).

New Protocols

Inside The Kernel

Implementing a new protocol first means deciding what the manipulatable and non-manipulatable parts of the tuple should be. Everything in the tuple has the property that it identifies the stream uniquely. The manipulatable part of the tuple is the part you can do NAT with: for TCP this is the source port, for ICMP it's the icmp ID; something to use as a "stream identifier". The non-manipulatable part is the rest of the packet that uniquely identifies the stream, but we can't play with (eg. TCP destination port, ICMP type).

Once you've decided this, you can write an extension to the connection-tracking code in the directory, and go about populating the `ip_conntrack_protocol' structure which you need to pass to `ip_conntrack_register_protocol()'.

The fields of `struct ip_conntrack_protocol' are:

list

Set it to '{ NULL, NULL }'; used to sew you into the list.

proto

Your protocol number; see `/etc/protocols'.

name

The name of your protocol. This is the name the user will see; it's usually best if it's the canonical name in `/etc/protocols'.

pkt_to_tuple

The function which fills out the protocol specific parts of the tuple, given the packet. The `datah' pointer points to the start of your header (just past the IP header), and the datalen is the length of the packet. If the packet isn't long enough to contain the header information, return 0; datalen will always be at least 8 bytes though (enforced by framework).

invert_tuple

This function is simply used to change the protocol-specific part of the tuple into the way a reply to that packet would look.

print_tuple

This function is used to print out the protocol-specific part of a tuple; usually it's sprintf()'d into the buffer provided. The number of buffer characters used is returned. This is used to print the states for the /proc entry.

print_conntrack

This function is used to print the private part of the conntrack structure, if any, also used for printing the states in /proc.

packet

This function is called when a packet is seen which is part of an established connection. You get a pointer to the conntrack structure, the IP header, the length, and the ctinfo. You return a verdict for the packet (usually NF_ACCEPT), or -1 if the packet is not a valid part of the connection. You can delete the connection inside this function if you wish, but you must use the following idiom to avoid races (see ip_conntrack_proto_icmp.c):

```
if (del_timer(&ct->timeout))
        ct->timeout.function((unsigned long)ct);
```

new

This function is called when a packet creates a connection for the first time; there is no ctinfo arg, since the first packet is of ctinfo IP_CT_NEW by definition. It returns 0 to fail to create the connection, or a connection timeout in jiffies.

Once you've written and tested that you can track your new protocol, it's time to teach NAT how to translate it. This means writing a new module; an extension to the NAT code and go about populating the `ip_nat_protocol' structure which you need to pass to `ip_nat_protocol_register()'.

list

Set it to '{ NULL, NULL }'; used to sew you into the list.

name

The name of your protocol. This is the name the user will see; it's best if it's the canonical name in `/etc/protocols' for userspace auto-loading, as we'll see later.

protonum

Your protocol number; see `/etc/protocols'.

manip_pkt

This is the other half of connection tracking's pkt_to_tuple function: you can think of it as "tuple_to_pkt". There are some differences though: you get a pointer to the start of the IP header, and the total packet length. This is because some protocols (UDP, TCP) need to know the IP header. You're given the ip_nat_tuple_manip field from the tuple (i.e., the "src" field), rather than the entire tuple, and the type of manipulation you are to perform.

in_range

This function is used to tell if manipulatable part of the given tuple is in the given range. This function is a bit tricky: we're given the manipulation type which has been applied to the tuple, which tells us how to interpret the range (is it a source range or a destination range we're aiming for?).

This function is used to check if an existing mapping puts us in the right range, and also to check if no manipulation is necessary at all.

unique_tuple

This function is the core of NAT: given a tuple and a range, we're to alter the per-protocol part of the tuple to place it within the range, and make it unique. If we can't find an unused tuple in the range, return 0. We also get a pointer to the conntrack structure, which is required for ip_nat_used_tuple().

The usual approach is to simply iterate the per-protocol part of the tuple through the range, checking `ip_nat_used_tuple()' on it, until one returns false.

Note that the null-mapping case has already been checked: it's either outside the range given, or already taken.

If IP_NAT_RANGE_PROTO_SPECIFIED isn't set, it means that the user is doing NAT, not NAPT: do something sensible with the range. If no mapping is desirable (for example, within TCP, a destination mapping should not change the TCP port unless ordered to), return 0.

print

Given a character buffer, a match tuple and a mask, write out the per-protocol parts and return the length of the buffer used.

print_range

Given a character buffer and a range, write out the per-protocol part of the range, and return the length of the buffer used. This won't be called if the IP_NAT_RANGE_PROTO_SPECIFIED flag wasn't set for the range.

New NAT Targets

This is the really interesting part. You can write new NAT targets which provide a new mapping type: two extra targets are provided in the default package: MASQUERADE and REDIRECT. These are fairly simple to illustrate the potential and power of writing a new NAT target.

These are written just like any other iptables targets, but internally they will extract the connection and call `ip_nat_setup_info()'.

Protocol Helpers

Protocol helpers for connection tracking allow the connection tracking code to understand protocols which use multiple network connections (eg. FTP) and mark the `child' connections as being related to the initial connection, usually by reading the related address out of the data stream.

Protocol helpers for NAT do two things: firstly allow the NAT code to manipulate the data stream to change the address contained within it, and secondly to perform NAT on the related connection when it comes in, based on the original connection.

Connection Tracking Helper Modules

Description

The duty of a connection tracking module is to specify which packets belong to an already established connection. The module has the following means to do that:

- Tell netfilter which packets our module is interested in (most helpers operate on a particular port).
- Register a function with netfilter. This function is called for every packet which matches the criteria above.
- An `ip_conntrack_expect_related()' function which can be called from there to tell netfilter to expect related connections.

If there is some additional work to be done at the time the first packet of the expected connection arrives, the module can register a callback function which is called at that time.

Structures and Functions Available

Your kernel module's init function has to call `ip_conntrack_helper_register()' with a pointer to a `struct ip_conntrack_helper'. This struct has the following fields:

list

This is the header for the linked list. Netfilter handles this list internally. Just initialize it with `{ NULL, NULL }'.

name

This is a pointer to a string constant specifying the name of the protocol. ("ftp", "irc", ...)

flags

A set of flags with one or more out of the following flgs:

- IP_CT_HELPER_F_REUSE_EXPECT
  Reuse expectations if the limit (see `max_expected` below) is reached.

me

A pointer to the module structure of the helper. Intitialize this with the `THIS_MODULE' macro.

max_expected

Maximum number of unconfirmed (outstanding) expectations.

timeout

Timeout (in seconds) for each unconfirmed expectation. An expectation is deleted `timeout' seconds after the expectation was issued with the `ip_conntrack_expect_related()' function.

tuple

This is a `struct ip_conntrack_tuple' which specifies the packets our conntrack helper module is interested in.

mask

Again a `struct ip_conntrack_tuple'. This mask specifies which bits of `tuple` are valid.

help

The function which netfilter should call for each packet matching tuple+mask

Example skeleton of a conntrack helper module

```
#define FOO_PORT        111

static int foo_expectfn(struct ip_conntrack *new)
{
        /* called when the first packet of an expected
           connection arrives */

        return 0;
}

static int foo_help(const struct iphdr *iph, size_t len,
                struct ip_conntrack *ct,
                enum ip_conntrack_info ctinfo)
{
        /* analyze the data passed on this connection and
           decide how related packets will look like */

        /* update per master-connection private data
           (session state, ...) */
        ct->help.ct_foo_info = ...

        if (there_will_be_new_packets_related_to_this_connection)
        {
                struct ip_conntrack_expect exp;

                memset(&exp, 0, sizeof(exp));
                exp.t = tuple_specifying_related_packets;
                exp.mask = mask_for_above_tuple;
                exp.expectfn = foo_expectfn;
                exp.seq = tcp_sequence_number_of_expectation_cause;

                /* per slave-connection private data */
                exp.help.exp_foo_info = ...

                ip_conntrack_expect_related(ct, &exp);
        }
        return NF_ACCEPT;
}

static struct ip_conntrack_helper foo;

static int __init init(void)
{
        memset(&foo, 0, sizeof(struct ip_conntrack_helper);
```

```
        foo.name = "foo";
        foo.flags = IP_CT_HELPER_F_REUSE_EXPECT;
        foo.me = THIS_MODULE;
        foo.max_expected = 1;   /* one expectation at a time */
        foo.timeout = 0;        /* expectation never expires */

        /* we are interested in all TCP packets with destport 111 */
        foo.tuple.dst.protonum = IPPROTO_TCP;
        foo.tuple.dst.u.tcp.port = htons(FOO_PORT);
        foo.mask.dst.protonum = 0xFFFF;
        foo.mask.dst.u.tcp.port = 0xFFFF;
        foo.help = foo_help;

        return ip_conntrack_helper_register(&foo);
}

static void __exit fini(void)
{
        ip_conntrack_helper_unregister(&foo);
}
```

NAT helper modules

Description

NAT helper modules do some application specific NAT handling. Usually this includes on-the-fly manipulation of data: think about the PORT command in FTP, where the client tells the server which IP/port to connect to. Therefor an FTP helper module must replace the IP/port after the PORT command in the FTP control connection.

If we are dealing with TCP, things get slightly more complicated. The reason is a possible change of the packet size (FTP example: the length of the string representing an IP/port tuple after the PORT command has changed). If we change the packet size, we have a syn/ack difference between left and right side of the NAT box. (i.e. if we had extended one packet by 4 octets, we have to add this offset to the TCP sequence number of each following packet).

Special NAT handling of all related packets is required, too. Take as example again FTP, where all incoming packets of the DATA connection have to be NATed to the IP/port given by the client with the PORT command on the control connection, rather than going through the normal table lookup.

- callback for the packet causing the related connection (foo_help)
- callback for all related packets (foo_nat_expected)

Structures and Functions Available

Your nat helper module's `init()' function calls `ip_nat_helper_register()' with a pointer to a `struct ip_nat_helper'. This struct has the following members:

list

Just again the list header for netfilters internal use. Initialize this with { NULL, NULL }.

name

A pointer to a string constant with the protocol's name

flags

A set out of zero, one or more of the following flags:

- IP_NAT_HELPER_F_ALWAYS
  Call the NAT helper for every packet, not only for packets where conntrack has detected an expectation-cause.
- IP_NAT_HELPER_F_STANDALONE
  Tell the NAT core that this protocol doesn't have a conntrack helper, only a NAT helper.

me

A pointer to the module structure of the helper. Initialize this using the `THIS_MODULE' macro.

tuple

a `struct ip_conntrack_tuple' describing which packets our NAT helper is interested in.

mask

a `struct ip_conntrack_tuple', telling netfilter which bits of `tuple` are valid.

help

The help function which is called for each packet matching tuple+mask.

expect

The expect function which is called for every first packet of an expected connection.

This is very similar to writing a connection tracking helper.

Example NAT helper module

```
#define FOO_PORT        111

static int foo_nat_expected(struct sk_buff **pksb,
                        unsigned int hooknum,
                        struct ip_conntrack *ct,
                        struct ip_nat_info *info)
/* called whenever the first packet of a related connection arrives.
   params:      pksb    packet buffer
                hooknum HOOK the call comes from (POST_ROUTING, PRE_ROUTING)
                ct      information about this (the related) connection
                info    &ct->nat.info
   return value: Verdict (NF_ACCEPT, ...)
{
        /* Change ip/port of the packet to the masqueraded
           values (read from master->tuplehash), to map it the same way,
           call ip_nat_setup_info, return NF_ACCEPT. */

}

static int foo_help(struct ip_conntrack *ct,
                    struct ip_conntrack_expect *exp,
                    struct ip_nat_info *info,
                    enum ip_conntrack_info ctinfo,
                    unsigned int hooknum,
                    struct sk_buff  **pksb)
/* called for every packet where conntrack detected an expectation-cause
   params:      ct      struct ip_conntrack of the master connection
                exp     struct ip_conntrack_expect of the expectation
                        caused by the conntrack helper for this protocol
                info    (STATE: related, new, established, ... )
                hooknum HOOK the call comes from (POST_ROUTING, PRE_ROUTING)
                pksb    packet buffer
*/
```

```
{

        /* extract information about future related packets (you can
           share information with the connection tracking's foo_help).
           Exchange address/port with masqueraded values, insert tuple
           about related packets */
}

static struct ip_nat_helper hlpr;

static int __init(void)
{
        int ret;

        memset(&hlpr, 0, sizeof(struct ip_nat_helper));
        hlpr.list = { NULL, NULL };
        hlpr.tuple.dst.protonum = IPPROTO_TCP;
        hlpr.tuple.dst.u.tcp.port = htons(FOO_PORT);
        hlpr.mask.dst.protonum = 0xFFFF;
        hlpr.mask.dst.u.tcp.port = 0xFFFF;
        hlpr.help = foo_help;
        hlpr.expect = foo_nat_expect;

        ret = ip_nat_helper_register(hlpr);

        return ret;
}

static void __exit(void)
{
        ip_nat_helper_unregister(&hlpr);
}
```

4.5 Understanding Netfilter

Netfilter is pretty simple, and is described fairly thoroughly in the previous sections. However, sometimes it's necessary to go beyond what the NAT or ip_tables infrastructure offers, or you may want to replace them entirely.

One important issue for netfilter (well, in the future) is caching. Each skb has an `nfcache' field: a bitmask of what fields in the header were examined, and whether the packet was altered or not. The idea is that each hook off netfilter OR's in the bits relevant to it, so that we can later write a cache system which will be clever enough to realize when packets do not need to be passed through netfilter at all.

The most important bits are NFC_ALTERED, meaning the packet was altered (this is already used for IPv4's NF_IP_LOCAL_OUT hook, to reroute altered packets), and NFC_UNKNOWN, which means caching should not be done because some property which cannot be expressed was examined. If in doubt, simply set the NFC_UNKNOWN flag on the skb's nfcache field inside your hook.

4.6 Writing New Netfilter Modules

Plugging Into Netfilter Hooks

To receive/mangle packets inside the kernel, you can simply write a module which registers a "netfilter hook". This is basically an expression of interest at some given point; the actual points are protocol-specific, and defined in protocol-specific netfilter headers, such as "netfilter_ipv4.h".

To register and unregister netfilter hooks, you use the functions `nf_register_hook' and `nf_unregister_hook'. These each take a pointer to a `struct nf_hook_ops', which you populate as follows:

list

Used to sew you into the linked list: set to '{ NULL, NULL }'

hook

The function which is called when a packet hits this hook point. Your function must return NF_ACCEPT, NF_DROP or NF_QUEUE. If NF_ACCEPT, the next hook attached to that point will be called. If NF_DROP, the packet is dropped. If NF_QUEUE, it's queued. You receive a pointer to an skb pointer, so you can entirely replace the skb if you wish.

flush

Currently unused: designed to pass on packet hits when the cache is flushed. May never be implemented: set it to NULL.

pf

The protocol family, eg, `PF_INET' for IPv4.

hooknum

The number of the hook you are interested in, eg `NF_IP_LOCAL_OUT'.

Processing Queued Packets

This interface is currently used by ip_queue; you can register to handle queued packets for a given protocol. This has similar semantics to registering for a hook, except you can block processing the packet, and you only see packets for which a hook has replied `NF_QUEUE'.

The two functions used to register interest in queued packets are `nf_register_queue_handler()' and `nf_unregister_queue_handler()'. The function you register will be called with the `void *' pointer you handed it to `nf_register_queue_handler()'.

If no-one is registered to handle a protocol, then returning NF_QUEUE is equivalent to returning NF_DROP.

Once you have registered interest in queued packets, they begin queueing. You can do whatever you want with them, but you must call `nf_reinject()' when you are finished with them (don't simply kfree_skb() them). When you reinject an skb, you hand it the skb, the `struct nf_info' which your queue handler was given, and a verdict: NF_DROP causes them to be dropped, NF_ACCEPT causes them to continue to iterate through the hooks, NF_QUEUE causes them to be queued again, and NF_REPEAT causes the hook which queued the packet to be consulted again (beware infinite loops).

You can look inside the `struct nf_info' to get auxiliary information about the packet, such as the interfaces and hook it was on.

Receiving Commands From Userspace

It is common for netfilter components to want to interact with userspace. The method for doing this is by using the setsockopt mechanism. Note that each protocol must be modified to call nf_setsockopt() for setsockopt numbers it doesn't understand (and nf_getsockopt() for getsockopt numbers), and so far only IPv4, IPv6 and DECnet have been modified.

Using a now-familiar technique, we register a `struct nf_sockopt_ops' using the nf_register_sockopt() call. The fields of this structure are as follows:

list

Used to sew it into the linked list: set to '{ NULL, NULL }'.

pf

The protocol family you handle, eg. PF_INET.

set_optmin

and

set_optmax

These specify the (exclusive) range of setsockopt numbers handled. Hence using 0 and 0 means you have no setsockopt numbers.

set

This is the function called when the user calls one of your setsockopts. You should check that they have NET_ADMIN capability within this function.

get_optmin

and

get_optmax

These specify the (exclusive) range of getsockopt numbers handled. Hence using 0 and 0 means you have no getsockopt numbers.

get

This is the function called when the user calls one of your getsockopts. You should check that they have NET_ADMIN capability within this function.

The final two fields are used internally.

4.7 Packet Handling in Userspace

Using the libipq library and the `ip_queue' module, almost anything which can be done inside the kernel can now be done in userspace. This means that, with some speed penalty, you can develop your code entirely in userspace. Unless you are trying to filter large bandwidths, you should find this approach superior to in-kernel packet mangling.

In the very early days of netfilter, I proved this by porting an embryonic version of iptables to userspace. Netfilter opens the doors for more people to write their own, fairly efficient netmangling modules, in whatever language they want.

Next Previous Contents