Linux netfilter Hacking HOWTO

$Revision: 521 $ $Date: 2002-07-02 06:07:19 +0200 (mar, 02 jul 2002) $

This document describes the netfilter architecture for Linux, how to hack it, and some of the major systems which sit on top of it, such as packet filtering, connection tracking and Network Address Translation.

IntroductionHi guys.This document is a journey; some parts are well-traveled, and in other areas you will find yourself almost alone. The best advice I can give you is to grab a large, cozy mug of coffee or hot chocolate, get into a comfortable chair, and absorb the contents before venturing out into the sometimes dangerous world of network hacking.For more understanding of the use of the infrastructure on top of the netfilter framework, I recommend reading the Packet Filtering HOWTO and the NAT HOWTO. For information on kernel programming I suggest Rusty's Unreliable Guide to Kernel Hacking and Rusty's Unreliable Guide to Kernel Locking.

1.

What is netfilter?netfilter is a framework for packet mangling, outside the normal Berkeley socket interface. It has four parts. Firstly, each protocol defines "hooks" (IPv4 defines 5) which are well-defined points in a packet's traversal of that protocol stack. At each of these points, the protocol will call the netfilter framework with the packet and the hook number.Secondly, parts of the kernel can register to listen to the different hooks for each protocol. So when a packet is passed to the netfilter framework, it checks to see if anyone has registered for that protocol and hook; if so, they each get a chance to examine (and possibly alter) the packet in order, then discard the packet (NF_DROP), allow it to pass (NF_ACCEPT), tell netfilter to forget about the packet (NF_STOLEN), or ask netfilter to queue the packet for userspace (NF_QUEUE).The third part is that packets that have been queued are collected (by the ip_queue driver) for sending to userspace; these packets are handled asynchronously.The final part consists of cool comments in the code and documentation. This is instrumental for any experimental project. The netfilter motto is (stolen shamelessly from Cort Dougan): “So... how is this better than KDE?”(This motto narrowly edged out 'Whip me, beat me, make me use ipchains').In addition to this raw framework, various modules have been written which provide functionality similar to previous (pre-netfilter) kernels, in particular, an extensible NAT system, and an extensible packet filtering system (iptables).

2.

What's wrong with what we had in 2.0 and 2.2?No infrastructure established for passing packet to userspace: Kernel coding is hardKernel coding must be done in C/C++Dynamic filtering policies do not belong in kernel 2.2 introduced copying packets to userspace via netlink, but reinjecting packets is slow, and subject to 'sanity' checks. For example, reinjecting packet claiming to come from an existing interface is not possible. Transparent proxying is a crock: We look up every packet to see if there is a socket bound to that address Root is allowed to bind to foreign addresses Can't redirect locally-generated packets REDIRECT doesn't handle UDP replies: redirecting UDP named packets to 1153 doesn't work because some clients don't like replies coming from anything other than port 53. REDIRECT doesn't coordinate with tcp/udp port allocation: a user may get a port shadowed by a REDIRECT rule. Has been broken at least twice during 2.1 series. Code is extremely intrusive. Consider the stats on the number of #ifdef CONFIG_IP_TRANSPARENT_PROXY in 2.2.1: 34 occurrences in 11 files. Compare this with CONFIG_IP_FIREWALL, which has 10 occurrences in 5 files. Creating packet filter rules independent of interface addresses is not possible: Must know local interface addresses to distinguish locally-generated or locally-terminating packets from through packets. Even that is not enough in cases of redirection or masquerading. Forward chain only has information on outgoing interface, meaning you have to figure where a packet came from using knowledge of the network topography. Masquerading is tacked onto packet filtering:Interactions between packet filtering and masquerading make firewalling complex: At input filtering, reply packets appear to be destined for box itselfAt forward filtering, demasqueraded packets are not seen at allAt output filtering, packets appear to come from local boxTOS manipulation, redirect, ICMP unreachable and mark (which can effect port forwarding, routing, and QoS) are tacked onto packet filter code as well. ipchains code is neither modular, nor extensible (eg. MAC address filtering, options filtering, etc). Lack of sufficient infrastructure has led to a profusion of different techniques: Masquerading, plus per-protocol modulesFast static NAT by routing code (doesn't have per-protocol handling)Port forwarding, redirect, auto forwardingThe Linux NAT and Virtual Server Projects. Incompatibility between CONFIG_NET_FASTROUTE and packet filtering: Forwarded packets traverse three chains anywayNo way to tell if these chains can be bypassed Inspection of packets dropped due to routing protection (eg. Source Address Verification) not possible. No way of atomically reading counters on packet filter rules. CONFIG_IP_ALWAYS_DEFRAG is a compile-time option, making life difficult for distributions who want one general-purpose kernel.

3.

Who are you?I'm the only one foolish enough to do this. As ipchains co-author and current Linux Kernel IP Firewall maintainer, I see many of the problems that people have with the current system, as well as getting exposure to what they are trying to do.

4.

Why does it crash?Woah! You should have seen it last week!Because I'm not as great a programmer as we might all wish, and I certainly haven't tested all scenarios, because of lack of time, equipment and/or inspiration. I do have a testsuite, which I encourage you to contribute to.

Where Can I Get The Latest?There is a CVS server on netfilter.org which contains the latest HOWTOs, userspace tools and testsuite. For casual browsing, you can use the . To grab the latest sources, you can do the following: Log in to the netfilter CVS server anonymously: cvs -d :pserver:cvs@pserver.netfilter.org:/cvspublic login When it asks you for a password type 'cvs'. Check out the code using: # cvs -d :pserver:cvs@pserver.netfilter.org:/cvspublic co netfilter/userspace To update to the latest version, use cvs update -d -PNetfilter ArchitectureNetfilter is merely a series of hooks in various points in a protocol stack (at this stage, IPv4, IPv6 and DECnet). The (idealized) IPv4 traversal diagram looks like the following:A Packet Traversing the Netfilter System: --->[1]--->[ROUTE]--->[3]--->[4]---> | ^ | | | [ROUTE] v | [2] [5] | ^ | | v |On the left is where packets come in: having passed the simple sanity checks (i.e., not truncated, IP checksum OK, not a promiscuous receive), they are passed to the netfilter framework's NF_IP_PRE_ROUTING [1] hook.Next they enter the routing code, which decides whether the packet is destined for another interface, or a local process. The routing code may drop packets that are unroutable.If it's destined for the box itself, the netfilter framework is called again for the NF_IP_LOCAL_IN [2] hook, before being passed to the process (if any).If it's destined to pass to another interface instead, the netfilter framework is called for the NF_IP_FORWARD [3] hook.The packet then passes a final netfilter hook, the NF_IP_POST_ROUTING [4] hook, before being put on the wire again.The NF_IP_LOCAL_OUT [5] hook is called for packets that are created locally. Here you can see that routing occurs after this hook is called: in fact, the routing code is called first (to figure out the source IP address and some IP options): if you want to alter the routing, you must alter the 'skb->dst' field yourself, as is done in the NAT code.

1.

Netfilter BaseNow we have an example of netfilter for IPv4, you can see when each hook is activated. This is the essence of netfilter.Kernel modules can register to listen at any of these hooks. A module that registers a function must specify the priority of the function within the hook; then when that netfilter hook is called from the core networking code, each module registered at that point is called in the order of priorites, and is free to manipulate the packet. The module can then tell netfilter to do one of five things: NF_ACCEPT: continue traversal as normal. NF_DROP: drop the packet; don't continue traversal. NF_STOLEN: I've taken over the packet; don't continue traversal. NF_QUEUE: queue the packet (usually for userspace handling). NF_REPEAT: call this hook again.The other parts of netfilter (handling queued packets, cool comments) will be covered in the kernel section later.Upon this foundation, we can build fairly complex packet manipulations, as shown in the next two sections.

2.

Packet Selection: IP TablesA packet selection system called IP Tables has been built over the netfilter framework. It is a direct descendent of ipchains (that came from ipfwadm, that came from BSD's ipfw IIRC), with extensibility. Kernel modules can register a new table, and ask for a packet to traverse a given table. This packet selection method is used for packet filtering (the 'filter' table), Network Address Translation (the 'nat' table) and general pre-route packet mangling (the 'mangle' table).The hooks that are registered with netfilter are as follows (with the functions in each hook in the order that they are actually called): --->PRE------>[ROUTE]--->FWD---------->POST------> Conntrack | Mangle ^ Mangle Mangle | Filter | NAT (Src) NAT (Dst) | | Conntrack (QDisc) | [ROUTE] v | IN Filter OUT Conntrack | Conntrack ^ Mangle | Mangle | NAT (Dst) v | Filter

2.1.

Packet FilteringThis table, 'filter', should never alter packets: only filter them.One of the advantages of iptables filter over ipchains is that it is small and fast, and it hooks into netfilter at the NF_IP_LOCAL_IN, NF_IP_FORWARD and NF_IP_LOCAL_OUT points. This means that for any given packet, there is one (and only one) possible place to filter it. This makes things much simpler for users than ipchains was. Also, the fact that the netfilter framework provides both the input and output interfaces for the NF_IP_FORWARD hook means that many kinds of filtering are far simpler.Note: I have ported the kernel portions of both ipchains and ipfwadm as modules on top of netfilter, enabling the use of the old ipfwadm and ipchains userspace tools without requiring an upgrade.

2.2.

NATThis is the realm of the 'nat' table, which is fed packets from two netfilter hooks: for non-local packets, the NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING hooks are perfect for destination and source alterations respectively. If CONFIG_IP_NF_NAT_LOCAL is defined, the hooks NF_IP_LOCAL_OUT and NF_IP_LOCAL_IN are used for altering the destination of local packets.This table is slightly different from the 'filter' table, in that only the first packet of a new connection will traverse the table: the result of this traversal is then applied to all future packets in the same connection.

2.2.1.

Masquerading, Port Forwarding, Transparent ProxyingI divide NAT into Source NAT (where the first packet has its source altered), and Destination NAT (the first packet has its destination altered).Masquerading is a special form of

Source NAT: port forwarding and transparent proxying are special forms of Destination NAT. These are now all done using the NAT framework, rather than being independent entities.

2.3.

Packet ManglingThe packet mangling table (the 'mangle' table) is used for actual changing of packet information. Example applications are the TOS and TCPMSS targets. The mangle table hooks into all five netfilter hooks. (please note this changed with kernel 2.4.18. Previous kernels didn't have mangle attached to all hooks)

3.

Connection TrackingConnection tracking is fundamental to NAT, but it is implemented as a separate module; this allows an extension to the packet filtering code to simply and cleanly use connection tracking (the 'state' module).

4.

Other AdditionsThe new flexibility provides both the opportunity to do really funky things, but for people to write enhancements or complete replacements that can be mixed and matched.

Information for ProgrammersI'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.

1.

Understanding ip_tablesiptables 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.

1.1.

ip_tables Data StructuresFor 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: A 'struct ipt_entry'. Zero or more 'struct ipt_entry_match' structures, each with a variable amount (0 or more bytes) of data appended to it. 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: A 'struct ipt_ip' part, containing the specifications for the IP header that it is to match. An 'nf_cache' bitfield showing what parts of the packet this rule examined. 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). 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. A ‘comefrom’ field used by the kernel to track packet traversal. 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.

1.2.

ip_tables From UserspaceUserspace 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’.

1.3.

ip_tables Use And TraversalThe 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.

2.

Extending iptablesBecause 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.

2.1.

The KernelWriting 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.

2.1.1.

New Match FunctionsNew 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:listThis field is set to any junk, say ‘{ NULL, NULL }’.nameThis 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.matchThis 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.checkentryThis 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.destroyThis 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.meThis 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.

2.1.2.

New TargetsIf 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:listThis field is set to any junk, say '{ NULL, NULL }'.nameThis 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.targetThis 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.).checkentryThis 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.destroyThis 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.meThis 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.

2.1.3.

New TablesYou 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:listThis field is set to any junk, say '{ NULL, NULL }'.nameThis 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.tableThis 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_hooksThis 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.lockThis is the read-write spinlock for the entire table; initialize it to RW_LOCK_UNLOCKED.privateThis is used internally by the ip_tables code.

2.2.

Userspace ToolNow 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.

2.2.1.

New Match FunctionsYour shared library's _init() function hands 'register_match()' a pointer to a static 'struct iptables_match', which has the following fields:nextThis pointer is used to make a linked list of matches (such as used for listing rules). It should be set to NULL initially.nameThe name of the match function. This should match the library name (eg "tcp" for 'libipt_tcp.so').versionUsually 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.sizeThe size of the match data for this match; you should use the IPT_ALIGN() macro to ensure it is correctly aligned.userspacesizeFor 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.helpA function which prints out the option synopsis.initThis 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()'.parseThis 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_checkThis 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.printThis 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.saveThis is the reverse of parse: it is used by 'iptables-save' to reproduce the options which created the rule.extra_optsThis 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.

2.2.2.

New TargetsYour 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.

2.3.

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.

3.

Understanding NATWelcome 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.

3.1.

Connection TrackingConnection 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_ESTABLISHEDThe packet is part of an established connection, in the original direction.IP_CT_RELATEDThe packet is related to the connection, and is passing in the original direction.IP_CT_NEWThe packet is trying to create a new connection (obviously, it is in the original direction).IP_CT_ESTABLISHED + IP_CT_IS_REPLYThe packet is part of an established connection, in the reply direction.IP_CT_RELATED + IP_CT_IS_REPLYThe 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.

Extending Connection Tracking/NATThese 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.

4.1.

Standard NAT TargetsNAT 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).

4.2.

New Protocols

4.2.1.

Inside The KernelImplementing 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:listSet it to '{ NULL, NULL }'; used to sew you into the list.protoYour protocol number; see '/etc/protocols'.nameThe 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_tupleThe 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_tupleThis function is simply used to change the protocol-specific part of the tuple into the way a reply to that packet would look.print_tupleThis 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_conntrackThis function is used to print the private part of the conntrack structure, if any, also used for printing the states in /proc.packetThis 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);newThis 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()'.listSet it to '{ NULL, NULL }'; used to sew you into the list.nameThe 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.protonumYour protocol number; see '/etc/protocols'.manip_pktThis 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_rangeThis 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_tupleThis 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.printGiven a character buffer, a match tuple and a mask, write out the per-protocol parts and return the length of the buffer used.print_rangeGiven 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.

4.3.

New NAT TargetsThis 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()’.

4.4.

Protocol HelpersProtocol 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.

4.5.

Connection Tracking Helper Modules

4.5.1.

DescriptionThe 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.

4.5.2.

Structures and Functions AvailableYour 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:listThis is the header for the linked list. Netfilter handles this list internally. Just initialize it with ‘{ NULL, NULL }’.nameThis is a pointer to a string constant specifying the name of the protocol. ("ftp", "irc", ...)flagsA 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.meA pointer to the module structure of the helper. Intitialize this with the ‘THIS_MODULE’ macro.max_expectedMaximum number of unconfirmed (outstanding) expectations.timeoutTimeout (in seconds) for each unconfirmed expectation. An expectation is deleted ‘timeout’ seconds after the expectation was issued with the ‘ip_conntrack_expect_related()’ function.tupleThis is a ‘struct ip_conntrack_tuple’ which specifies the packets our conntrack helper module is interested in.maskAgain a ‘struct ip_conntrack_tuple’. This mask specifies which bits of tuple are valid.helpThe function which netfilter should call for each packet matching tuple+mask

4.5.3.

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); }

4.6.

NAT helper modules

4.6.1.

DescriptionNAT 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)

4.6.2.

Structures and Functions AvailableYour 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:listJust again the list header for netfilters internal use. Initialize this with { NULL, NULL }.nameA pointer to a string constant with the protocol's nameflagsA 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.meA pointer to the module structure of the helper. Initialize this using the 'THIS_MODULE' macro.tuplea 'struct ip_conntrack_tuple' describing which packets our NAT helper is interested in.maska 'struct ip_conntrack_tuple', telling netfilter which bits of tuple are valid.helpThe help function which is called for each packet matching tuple+mask.expectThe expect function which is called for every first packet of an expected connection.This is very similar to writing a connection tracking helper.

4.6.3.

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); }

5.

Understanding NetfilterNetfilter 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.

6.

Writing New Netfilter Modules

6.1.

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:listUsed to sew you into the linked list: set to '{ NULL, NULL }'hookThe 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.flushCurrently unused: designed to pass on packet hits when the cache is flushed. May never be implemented: set it to NULL.pfThe protocol family, eg, 'PF_INET' for IPv4.hooknumThe number of the hook you are interested in, eg 'NF_IP_LOCAL_OUT'.

6.2.

Processing Queued PacketsThis 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.

6.3.

Receiving Commands From UserspaceIt 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:listUsed to sew it into the linked list: set to ’{ NULL, NULL }’.pfThe protocol family you handle, eg. PF_INET.set_optminandset_optmaxThese specify the (exclusive) range of setsockopt numbers handled. Hence using 0 and 0 means you have no setsockopt numbers.setThis 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_optminandget_optmaxThese specify the (exclusive) range of getsockopt numbers handled. Hence using 0 and 0 means you have no getsockopt numbers.getThis 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.

7.

Packet Handling in UserspaceUsing 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.

Translating 2.0 and 2.2 Packet Filter ModulesLook at the ip_fw_compat.c file for a simple layer which should make porting quite simple.Netfilter Hooks for Tunnel WritersAuthors of tunnel (or encapsulation) drivers should follow two simple rules for the 2.4 kernel (as do the drivers inside the kernel, like net/ipv4/ipip.c):Release skb->nfct if you’re going to make the packet unrecognisable (ie. decapsulating/encapsulating). You don’t need to do this if you unwrap it into a *new* skb, but if you’re going to do it in place, you must do this. Otherwise: the NAT code will use the old connection tracking information to mangle the packet, with bad consequences.Make sure the encapsulated packets go through the LOCAL_OUT hook, and decapsulated packets go through the PRE_ROUTING hook (most tunnels use ip_rcv(), which does this for you). Otherwise: the user will not be able to filter as they expect to with tunnels.The canonical way to do the first is to insert code like the following before you wrap or unwrap the packet: /* Tell the netfilter framework that this packet is not the same as the one before! */ #ifdef CONFIG_NETFILTER nf_conntrack_put(skb->nfct); skb->nfct = NULL; #ifdef CONFIG_NETFILTER_DEBUG skb->nf_debug = 0; #endif #endifUsually, all you need to do for the second, is to find where the newly encapsulated packet goes into "ip_send()", and replace it with something like: /* Send "new" packet from local host */ NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev, ip_send); Following these rules means that the person setting up the packet filtering rules on the tunnel box will see something like the following sequence for a packet being tunnelled: FORWARD hook: normal packet (from eth0 -> tunl0) LOCAL_OUT hook: encapsulated packet (to eth1).And for the reply packet: LOCAL_IN hook: encapsulated reply packet (from eth1) FORWARD hook: reply packet (from eth1 -> eth0).The Test SuiteWithin the CVS repository lives a test suite: the more the test suite covers, the greater confidence you can have that changes to the code hasn't quietly broken something. Trivial tests are at least as important as tricky tests: it's the trivial tests which simplify the complex tests (since you know the basics work fine before the complex test gets run).The tests are simple: they are just shell scripts under the testsuite/ subdirectory which are supposed to succeed. The scripts are run in alphabetical order, so '01test' is run before '02test'. Currently there are 5 test directories:00netfilter/General netfilter framework tests.01iptables/iptables tests.02conntrack/connection tracking tests.03NAT/NAT tests04ipchains-compat/ipchains/ipfwadm compatibility testsInside the testsuite/ directory is a script called 'test.sh'. It configures two dummy interfaces (tap0 and tap1), turns forwarding on, and removes all netfilter modules. Then it runs through the directories above and runs each of their test.sh scripts until one fails. This script takes two optional arguments: '-v' meaning to print out each test as it proceeds, and an optional test name: if this is given, it will skip over all tests until this one is found.

1.

Writing a TestCreate a new file in the appropriate directory: try to number your test so that it gets run at the right time. For example, in order to test ICMP reply tracking (02conntrack/02reply.sh), we need to first check that outgoing ICMPs are tracked properly (02conntrack/01simple.sh).It's usually better to create many small files, each of which covers one area, because it helps to isolate problems immediately for people running the testsuite.If something goes wrong in the test, simply do an 'exit 1', which causes failure; if it's something you expect may fail, you should print a unique message. Your test should end with 'exit 0' if everything goes OK. You should check the success of every command, either using 'set -e' at the top of the script, or appending '|| exit 1' to the end of each command.The helper functions 'load_module' and 'remove_module' can be used to load modules: you should never rely on autoloading in the testsuite unless that is what you are specifically testing.

2.

Variables And EnvironmentYou have two play interfaces: tap0 and tap1. Their interface addresses are in variables $TAP0 and $TAP1 respectively. They both have netmasks of 255.255.255.0; their networks are in $TAP0NET and $TAP1NET respectively.There is an empty temporary file in $TMPFILE. It is deleted at the end of your test.Your script will be run from the testsuite/ directory, wherever it is. Hence you should access tools (such as iptables) using path starting with '../userspace'.Your script can print out more information if $VERBOSE is set (meaning that the user specified '-v' on the command line).

3.

Useful ToolsThere are several useful testsuite tools in the "tools" subdirectory: each one exits with a non-zero exit status if there is a problem.

3.1.

gen_ipYou can generate IP packets using ‘gen_ip’, which outputs an IP packet to standard input. You can feed packets in the tap0 and tap1 by sending standard output to /dev/tap0 and /dev/tap1 (these are created upon first running the testsuite if they don’t exist).gen_ip is a simplistic program which is currently very fussy about its argument order. First are the general optional arguments:FRAG=offset,lengthGenerate the packet, then turn it into a fragment at the following offset and length.MFSet the ‘More Fragments’ bit on the packet.MAC=xx:xx:xx:xx:xx:xxSet the source MAC address on the packet.TOS=tosSet the TOS field on the packet (0 to 255).Next come the compulsory arguments:source ipSource IP address of the packet.dest ipDestination IP address of the packet.lengthTotal length of the packet, including headers.protocolProtocol number of the packet, eg 17 = UDP.Then the arguments depend on the protocol: for UDP (17), they are the source and destination port numbers. For ICMP (1), they are the type and code of the ICMP message: if the type is 0 or 8 (ping-reply or ping), then two additional arguments (the ID and sequence fields) are required. For TCP, the source and destination ports, and flags ("SYN", "SYN/ACK", "ACK", "RST" or "FIN") are required. There are three optional arguments: "OPT=" followed by a comma-separated list of options, "SYN=" followed by a sequence number, and "ACK=" followed by a sequence number. Finally, the optional argument "DATA" indicates that the payload of the TCP packet is to be filled with the contents of standard input.

3.2.

rcv_ipYou can see IP packets using ‘rcv_ip’, which prints out the command line as close as possible to the original value fed to gen_ip (fragments are the exception).This is extremely useful for analyzing packets. It takes two compulsory arguments:wait timeThe maximum time in seconds to wait for a packet from standard input.iterationsThe number of packets to receive.There is one optional argument, "DATA", which causes the payload of a TCP packet to be printed on standard output after the packet header.The standard way to use ‘rcv_ip’ in a shell script is as follows:# Set up job control, so we can use & in shell scripts. set -m # Wait two seconds for one packet from tap0 ../tools/rcv_ip 2 1 < /dev/tap0 > $TMPFILE & # Make sure that rcv_ip has started running. sleep 1 # Send a ping packet ../tools/gen_ip $TAP1NET.2 $TAP0NET.2 100 1 8 0 55 57 > /dev/tap1 || exit 1 # Wait for rcv_ip, if wait %../tools/rcv_ip; then : else echo rcv_ip failed: cat $TMPFILE exit 1 fi

3.3.

gen_errThis program takes a packet (as generated by gen_ip, for example) on standard input, and turns it into an ICMP error.It takes three arguments: a source IP address, a type and a code. The destination IP address will be set to the source IP address of the packet fed in standard input.

3.4.

local_ipThis takes a packet from standard input and injects it into the system from a raw socket. This give the appearance of a locally-generated packet (as separate from feeding a packet in one of the ethertap devices, which looks like a remotely-generated packet).

4.

Random AdviceAll the tools assume they can do everything in one read or write: this is true for the ethertap devices, but might not be true if you're doing something tricky with pipes.dd can be used to cut packets: dd has an obs (output block size) option which can be used to make it output the packet in a single write.Test for success first: eg. testing that packets are successfully blocked. First test that packets pass through normally, then test that some packets are blocked. Otherwise an unrelated failure could be stopping the packets...Try to write exact tests, not 'throw random stuff and see what happens' tests. If an exact test goes wrong, it's a useful thing to know. If a random test goes wrong once, it doesn't help much.If a test fails without a message, you can add '-x' to the top line of the script (ie. '#! /bin/sh -x') to see what commands it's running.If a test fails randomly, check for random network traffic interfering (try downing all your external interfaces). Sitting on the same network as Andrew Tridgell, I tend to get plagued by Windows broadcasts, for example.

MotivationAs I was developing ipchains, I realized (in one of those blinding-flash-while-waiting-for-entree moments in a Chinese restaurant in Sydney) that packet filtering was being done in the wrong place. I can't find it now, but I remember sending mail to Alan Cox, who kind of said 'why don't you finish what you're doing, first, even though you're probably right'. In the short term, pragmatism was to win over The Right Thing.After I finished ipchains, which was initially going to be a minor modification of the kernel part of ipfwadm, and turned into a larger rewrite, and wrote the HOWTO, I became aware of just how much confusion there is in the wider Linux community about issues like packet filtering, masquerading, port forwarding and the like.This is the joy of doing your own support: you get a closer feel for what the users are trying to do, and what they are struggling with. Free software is most rewarding when it's in the hands of the most users (that's the point, right?), and that means making it easy. The architecture, not the documentation, was the key flaw.So I had the experience, with the ipchains code, and a good idea of what people out there were doing. There were only two problems.Firstly, I didn't want to get back into security. Being a security consultant is a constant moral tug-of-war between your conscience and your wallet. At a fundamental level, you are selling the feeling of security, which is at odds with actual security. Maybe working in a military setting, where they understand security, it'd be different.The second problem is that newbie users aren't the only concern; an increasing number of large companies and ISPs are using this stuff. I needed reliable input from that class of users if it was to scale to tomorrow's home users.These problems were resolved, when I ran into David Bonn, of WatchGuard fame, at Usenix in July 1998. They were looking for a Linux kernel coder; in the end we agreed that I'd head across to their Seattle offices for a month and we'd see if we could hammer out an agreement whereby they'd sponsor my new code, and my current support efforts. The rate we agreed on was more than I asked, so I didn't take a pay cut. This means I don't have to even think about external conslutting for a while.Exposure to WatchGuard gave me exposure to the large clients I need, and being independent from them allowed me to support all users (eg. WatchGuard competitors) equally.So I could have simply written netfilter, ported ipchains over the top, and been done with it. Unfortunately, that would leave all the masquerading code in the kernel: making masquerading independent from filtering is the one of the major wins point of moving the packet filtering points, but to do that masquerading also needed to be moved over to the netfilter framework as well.Also, my experience with ipfwadm's 'interface-address' feature (the one I removed in ipchains) had taught me that there was no hope of simply ripping out the masquerading code and expecting someone who needed it to do the work of porting it onto netfilter for me.So I needed to have at least as many features as the current code; preferably a few more, to encourage niche users to become early adopters. This means replacing transparent proxying (gladly!), masquerading and port forwarding. In other words, a complete NAT layer.Even if I had decided to port the existing masquerading layer, instead of writing a generic NAT system, the masquerading code was showing its age, and lack of maintenance. See, there was no masquerading maintainer, and it shows. It seems that serious users generally don't use masquerading, and there aren't many home users up to the task of doing maintenance. Brave people like Juan Ciarlante were doing fixes, but it had reached to the stage (being extended over and over) that a rewrite was needed.Please note that I wasn't the person to do a NAT rewrite: I didn't use masquerading any more, and I'd not studied the existing code at the time. That's probably why it took me longer than it should have. But the result is fairly good, in my opinion, and I sure as hell learned a lot. No doubt the second version will be even better, once we see how people use it.ThanksThanks to those who helped, expecially Harald Welte for writing the Protocol Helpers section.