Netfilter Connection Tracking and NAT Implementation

Magnus Boye
Aalto University School of Electrical Engineering
Department of Communications and Networking
P.O.Box 13000, 00076 Aalto, Finland
Email: firstname.lastname@aalto.fi

ABSTRACT

Good sources of information about the implementation of the Linux kernel are scarce. Due to the constant development, existing documentation quickly becomes outdated, although the general architecture of the kernel rarely changes radically. This paper attempts to give a detailed overview of the connection tracking and NAT modules in Netfilter. Understanding the architecture and implementation of these modules is necessary in order to modify or extend Netfilter. The architecture and implementation covered in this paper are based on kernel version 3.5.4.

Keywords

Linux kernel, Netfilter, connection tracking, NAT

1. INTRODUCTION

The small address space of IPv4 inevitably caused Network Address Translation (NAT) to be used in networks that are not assigned a public IP address range. In reality this is mostly residential Internet gateways where NAT offers multiple devices to share a single public IP address. It can be argued that NAT provides some level of security by hiding the structure of the LAN connected to the gateway. However, NAT is generally disliked in the networking community because it breaks the end-to-end principle, and IPv6 offers a solution to the problem that caused NAT to be used in the first place.

NAT is a stateful system that keeps track of incoming and outgoing flows of a network. NAT ensures that outgoing flows are mapped to a unique combination of IP address and transport-layer identifier on the external network. Figure 1 shows an illustration of source and destination NAT. Two client hosts A and B connect through a NAT gateway to an external network and have coincidentally selected the same source port for their outgoing flows. The NAT gateway performs source NAT on the outgoing flows by replacing the source IP address with the address of the gateway on the external network, and the source ports with available source ports associated with the IP address on the external network. Source NAT guarantees that a combination of internal network IP address and source port is mapped to a unique combination of external network IP address and source port. If such a mapping is not possible, traffic is dropped. When return traffic arrives at the NAT gateway, the destination IP address and destination port is replaced with the original source IP address and port. The gateway also performs destination NAT which is commonly called "port forwarding" in consumer routers. In figure 1, the gateway performs destination NAT on port 80 and 22 on incoming traffic from the external network. Destination NAT works by modifying the destination of incoming traffic, just like source NAT does for return traffic. In this scenario traffic to port 80 and 22 are directed to the hosts WWW and SSH on the internal network, respectively. Source NAT and destination NAT can be used simultaneously, as long as source NAT does not map any outgoing flows to ports used by destination NAT. Depending on the configuration, destination NAT can have various purposes, for instance load balancing.

In addition to transport-layer protocols, some protocols such as ICMP also use identifiers to distinguish between flows. The most common identifiers are 16 bit port numbers as used by TCP and UDP. The size of the protocol-specific identifier determines the number of connections the gateway can handle for the protocol. A 16 bit identifier theoretically allows for up to $2^{16}-1$ simultaneous flows through a gateway. The number of possible simultaneous flows can be increased by using NAT with multiple external IP addresses.

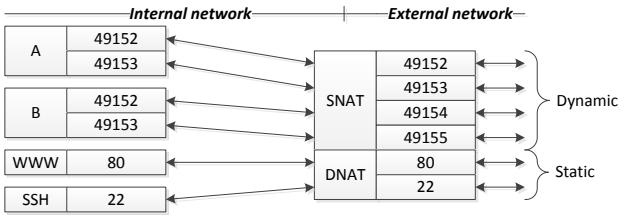

Figure 1: Source and destination NAT.

The book by K. Wehrle et al.[4] gives a broad and detailed overview of the Linux networking architecture and implementation. The book describes the architecture as of kernel version 2.4 which dates back to 2001. Many aspects of the architecture described in the book are similar with the kernel version 3.5.4, but the source code is far from the same. J. Engelhardt[1] is maintaining a short guide on how to create Netfilter modules. The guide gives a brief introduction to Netfilter and connection tracking, but does not describe the architecture and implementation of connection tracking and NAT in detail. The connection tracking and NAT modules support several transport-layer protocols. In order to give a more concise overview in this paper, it was decided to focus on the generic aspects of the modules, and UDP over IPv4.

This paper is structured as follows. Section 2 gives a short introduction to the Netfilter hooks used Netfilter modules. Section 3 gives an overview of the connection tracking module, its key data structure and functions. Section 4 gives an overview of the NAT module and how it relates to the connection tracking module. Section 5 describes potential vulnerabilities in the implementations of connection tracking and NAT.

2. NETFILTER FRAMEWORK

Netfilter is a framework for packet manipulation and filtering. The framework provides access to packets through five hooks in the Linux kernel at key points in packet processing. The hooks exist for both IPv4 and IPv6. Figure 2 shows in which order the Netfilter hooks are called when processing an IPv4 packet. The return value from a Netfilter hook must be one of five options: `NF_ACCEPT`, `NF_DROP`, `NF_STOLEN`, `NF_QUEUE`, `NF_REPEAT`. The return value is also referred to as a *verdict*. The first two options accept or drop a packet, respectively. If multiple functions are attached to a hook, the packet will be dropped if a single function returns `NF_DROP`. The return value `NF_STOLEN` indicates that a packet has been consumed by the hook function and further processing by other functions attached to the hook is not possible. `NF_QUEUE` indicates that the packet should be inserted into a queue, and `NF_REPEAT` indicates that the hook function should be called again. When functions are registered to hooks, a priority of the functions is given. This priority determines the order order in which the functions attached to the same hook are called. This paper focuses on events that mainly take place at the `NF_IP_PRE_ROUTING` and `NF_IP_POST_ROUTING` hooks, as these are key points in connection tracking and NAT.

Hooks are registered by calling `nf_register_hook()`, which takes a pointer to a `struct nf_hook_ops` as parameter. This structure defines the actual function called by the hook, address family, priority, and at which hook the function should be called as shows in Figure 2. The various hooks are integrated into the IPv4 implementation using two functions: `NF_HOOK()` and `NF_HOOK_COND()`. The former executes hooks and the later works just like `NF_HOOK`, except an execution condition can be given. Both functions are passed a pointer to `okfn()` which is the callback function for the hook point. The provided `okfn()` is called if the condition is `false` when using `NF_HOOK_COND()`, or if the final verdict of the functions registered to the hook is `NF_ACCEPT`. The functions in the IPv4 implementation where hooks are executed are shown in Table 1.

Figure 3 shows how functions registered to hooks are executed when `NF_HOOK()` is called. `nf_hook_thresh()` check if Netfilter hooks are enabled and then calls `nf_hook_slow()`, which calls the main function `nf_iterate()` and evaluates the return value of this function. As the name suggests, `nf_iterate()` iterates over the function registered to the hook specified when `NF_HOOK()` was called. `nf_iterate()` calls each registered function and checks their return values. If any of the functions return `NF_DROP`, the packet must be dropped and thus it is not necessary to call any remaining functions for the hook. If the function return `NF_REPEAT`, the function is called again in `nf_iterate()`, and if no functions have been registered for the hook `NF_ACCEPT` is returned. The return value of `nf_iterate()` is evaluated by `nf_hook_slow()`, mainly to perform potential queuing. Only the eight least significant bits of the return value are used to store one of the five possible Netfilter verdicts. The remaining bits in the 32 bit return value can be used for other data. If the Netfilter verdict is `NF_QUEUE`, the 16 most significant bits are used to indicate which Netfilter queue the packet should be inserted into. The queue number is passed to `nf_queue()` which queues the packet. When the packet has passed through the queue it will be re-injected into the packet processing flow by the function `nf_reinject()`. If the verdict returned by `nf_iterate()` is `NF_DROP`, the function `kfree_skb()` is called and the packet is dropped.

Hook	Caller
NF_INET_PRE_ROUTING	ip_rcv()
NF_INET_LOCAL_IN	ip_local_deliver()
NF_INET_FORWARD	ip_forward()
NF_INET_LOCAL_OUT	__ip_local_out()
NF_INET_POST_ROUTING	ip_output()

Table 1: Netfilter hooks in IPv4.

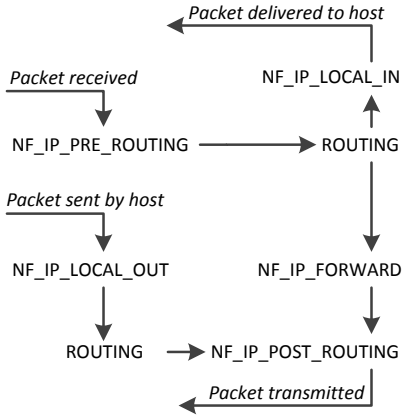

Figure 2: Netfilter IPv4 hook traversal.

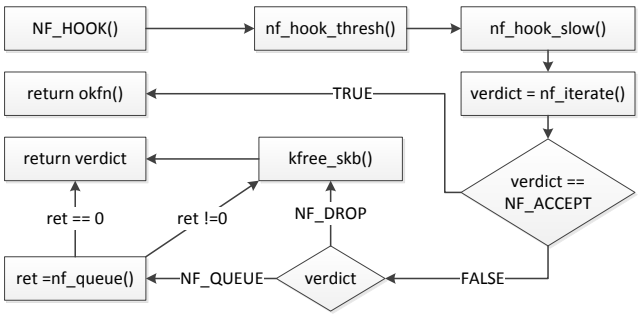

Figure 3: Hook execution by `nf_hook()`

3. CONNECTION TRACKING

The connection tracking(CT) module is responsible for identifying trackable packets belonging to trackable protocols. The module supports tracking of both stateless and statefull protocols. The CT module operates independently of the NAT module, but its primary purpose is to support the NAT module.

3.1 Tuples

The most important data structure of the CT module is `struct nf_conntrack_tuple`. This "tuple" structure is used to represent a unidirectional packet flow by its network-layer and transport-layer addresses. Bidirectional flows are thus represented using a tuple for each direction. Figure 4 shows a simplified representation of `struct nf_conntrack_tuple`. The data structure uses `unions` to contain both protocol-specific fields and generic fields in `dst.u`. This makes the source code easier to understand, optimizes memory, and allows new protocol-specific fields to be added without breaking the existing code. The `dst.u` field defines a `union` of 16 bit that contains fields for the following protocols: TCP, UDP, ICMP, DCCP, SCTP, and GRE. The fields reveal information about the header information in different protocols used to uniquely identify a packet flow. For instance, TCP and UDP use port numbers while ICMP uses ICMP type and code.

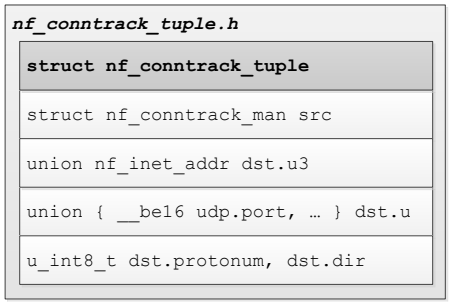

Figure 4: `struct nf_conntrack_tuple`

Since a tuple contains different information depending on both the network-layer protocol and transport-layer protocol of a packet, each supported protocol is implemented as a module. The modules conform to the interface defined by the two structures `struct nf_conntrack_l3proto` and `struct nf_conntrack_l4proto`. The structures contain function pointers that are initialized to the appropriate functions in the protocol-specific modules. Figure 5 shows the initialization values of the two structures for a UDP packet encapsulated by IPv4. The most important function pointer that both structures have in common is `pkt_to_tuple()`. This pointer points to a function which maps a packet to a tuple based on its network-layer or transport-layer data. In the case of IP, `pkt_to_tuple()` sets the `dst.u3` and `src.u3` fields of a tuple to the source and destination IP address of the packet, respectively. In the case of UDP, `pkt_to_tuple()` sets the `dst.u` and `src.u` fields to the source and destination UDP ports, respectively.

The `l3proto` and `l4proto` fields in Figure 5 are set to address family and protocol numbers as defined in the Linux kernel. Note that these values are not the same as specified by IANA[2][3], although some of them overlap. An explanation of the difference between Linux and IANA address families could not be found. Considering that IP was invented before IANA was founded, it is possible that the numbering of address families in Linux is simply a legacy from previous operating systems. The `get_timeouts()` function returns the timeout values of the protocol. The `error()` function checks for special packets that cannot be tracked, and `new()` is called when a new flow is seen by the CT module. Finally, `packet()` function is called for all packets which are deemed trackable by `error()`.

Constant	Description
`IPS_EXPECTED`	The connection was expected
`IPS_SEEN_REPLY`	Bidirectional traffic has been seen
`IPS_ASSURED`	Never expire connection prematurely
`IPS_CONFIRMED`	Packets were transmitted
`IPS_SEQ_ADJUST`	TCP needs sequence number adjustment
`IPS_DYING`	Connection is dying

Table 2: Connection status values.

3.2 Hashing

The CT module is optimized for performance and therefore stores the CT state of active connections in a hash table. The function `hash_conntrack_raw()` returns a generic 32 bit hash of a tuple. The hash value is based on the source and destination IP addresses and protocol-specific identifier. The `nf_conntrack_tuple_hash` structure is used to store a CT state in the hash table and contains the tuple along with a pointer to a linked list of CT state associated with the tuple. The linked list is used to handle hash collisions.

3.3 Connections

Netfilter uses the term *connection* even for packet flows in connectionless protocols. For the sake of clarity the term flow is used in this paper. A tracked flow is represented by a `struct nf_conn` which is shown in Figure 6. The `tuplehash` field contains a `struct nf_conntrack_tuple_hash` for each direction of the flow, and these structures contain a reverse pointer to the `nf_conn` structure. The key fields in the data structure are `timeout` and `status`. The `timeout` field contains a list of timers related to the connection state. These timers are necessary in order to optimize resource utilization. The CT states of inactive flows are removed to reduce memory and enable faster hash table lookups. Connection-oriented protocols have multiple states and the lifetime of CT states for such flows depend on the protocol state. The connection tracking module gives priority to established connections in the case of connection-oriented protocols and bidirectional flows in the case of connection-less protocols. For instance, the CT state of a TCP connection has a lower lifetime during the initial three-way handshake compared to after the handshake has been completed. In the case of UDP, unidirectional flows have a shorter lifetime than bidirectional flows, because the bidirectionality indicates that a flow is important. The `status` field is used as a bitset where different bits correspond to different protocol states, as specified by `enum ip_conntrack_status`. The most important connection states are shown in Table 2.

3.4 Tracking

The CT modules uses three Netfilter hooks to track incoming and outgoing packets. The function `nf_conntrack_in()` is called by the `NF_INET_PRE_ROUTING` and `NF_INET_LOCAL_OUT` hooks. The function `nf_conntrack_confirm` is called by the `NF_INET_POST_ROUTING` hook. The `nf_conntrack_in()` function is the main function of the CT module. The initial steps of the `nf_conntrack_in()` function is to determine the

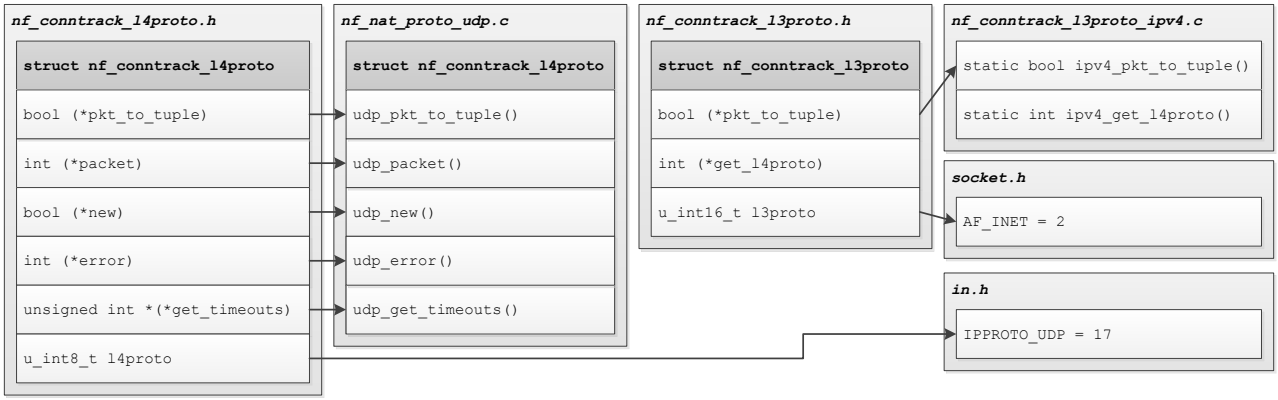

Figure 5: `struct nf_conntrack_l3proto` **and** `struct nf_conntrack_l4proto`

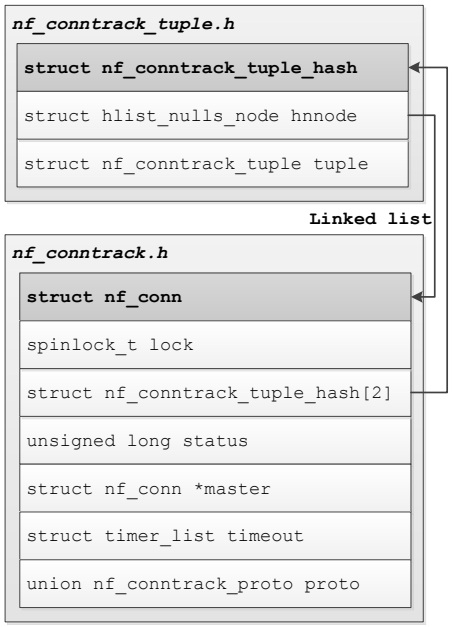

Figure 6: `struct nf_conntrack_tuple_hash` **and** `struct nf_conn`

network-layer protocol and transport-layer protocols. If the protocols can be tracked, a `struct nf_conntrack_l3proto` and a `struct nf_conntrack_l4proto` are initialized, as previously described. Before the main protocol-specific tracking functions are called, the `error()` function is called. In the case of UDP, the function checks for malformed packets with invalid payload size or invalid checksum. If the `error()` function returns `NF_ACCEPT` the packet is trackable and `resolve_normal_ct()` is called. This function ensures that a CT state exists for the packet tuple, by creating a new CT state if this is the first packet in a flow. The function begins by calling the protocol-specific `pkt_to_tuple()` function and obtains a tuple. The hash of the tuple or inverse tuple is calculated and used to retrieve a possibly existing CT state from the hash table of the CT module. If no CT state is found a new state is created by calling `init_conntrack()`. Otherwise the existing the CT state is returned.

The `init_conntrack()` function creates a new `nf_conn` structure and initializes its values by calling the protocol-specific function `new()`. The function continues by checking if the packet was expected as a result of protocol behavior in a flow tracked by another CT state. If the flow was expected, the `master` field of the `nf_conn` structure is set accordingly. Some application protocols utilizes multiple packet flows and in order for such applications to function properly, the NAT module must be able to direct incoming flows to the right host. Since protocol behavior varies from protocol to protocol, protocol-specific modules are needed to detect protocol behavior that results in new flows. For instance, in passive FTP the client tells the server to connect to the client on a specified address and port when a file transfer is requested. The CT module creates an *expected* CT state for the incoming flow, so that it can be identified and directed by the NAT module. The `struct nf_conntrack_tuple_hash` of the packet is inserted into a list of *unconfirmed* connections. If the packet is not dropped by any other Netfilter modules, the packet should be observed by `nf_conntrack_confirm` at the `NF_INET_POST_ROUTING` hook. The purpose of this function is to check that a packet belonging to a connection actually makes it onto the network and was not dropped by another module. If the packet is seen by this function, the state of the connection is changed to `IPS_CONFIRMED` and the connection is removed from the list of unconfirmed connections and inserted into the hash table of the CT module. After the `resolve_normal_ct()` function has ensured that a CT state exists, the function returns a pointer to the CT state of the packet.

`nf_conntrack_in()` continues by obtaining the protocol-specific timeout values and then calls the `packet()` which points to `udp_packet()` for the UDP protocol. Because UDP is connectionless, the connection tracking functions are not very advanced. The `udp_packet()` function simply extends the timeout of the connection based on whether the `IPS_SEEN_REPLY` bit has been set in the connection status. If bidirectional traffic has been seen, the connection timeout is extended further than if only unidirectional traffic has been seen. As mentioned earlier, the reason for this behavior is that the CT module optimizes resource consumption by inactive flows. The shorter timeout for unidirectional connections does not limit connectivity, but requires more frequent packet transmissions to prevent the flow from expiring. The

shorter timeout also makes the CT module more tolerant to denial-of-service attacks, although the default timeouts are still too high to mitigate attacks. The default timeout specified for UDP in the CT module is 30 seconds for a unidirectional (unreplied) connection, and 3 minutes for a bidirectional connection.

The `udp_packet()` function always returns `NF_ACCEPT` since screening for bad packets has already been performed by `udp_error()` and therefore nothing can go wrong in this function. The final verdict returned by `nf_conntrack_in()` is determined by `udp_error()` or `udp_packet()` function in the case of UDP.

4. NETWORK ADDRESS TRANSLATION

Although the Netfilter NAT module does modify the network-layer addresses of packets, it appears the addresses may be altered by other modules invoked after the NAT module. In particular the Masquerade module replaces the source IP address of packets such that they match that of the transmission interface. The module creates a NAT rule such that traffic in the reverse direction will have their destination address replaced with the original source address. This section covers transport-layer NAT. The inner workings of the Masquerade module and network-layer NAT is an area that needs further documentation.

The NAT module is comprised of a set of core functions that perform general NAT tasks, and several protocol-specific NAT modules. The main function of the NAT module is `nf_nat_fn()` which is called by four helper functions at the following Netfilter hooks: `NF_IP_PRE_ROUTING`, `NF_IP_POST_ROUTING`, `NF_IP_LOCAL_OUT`, and `NF_IP_LOCAL_IN`. The protocol-specific modules are needed because some protocols exchange address information at the application layer and this information needs to be altered according to the address modifications performed at the network and transport layers. The protocol-specific modules conform to the interface defined by `struct nf_nat_protocol`. The structure defines four function pointers and their relationship to the UDP NAT module is shown in Figure 7. The function `manip_pkt()` alters a packet based on a tuple and the type of NAT: source NAT or destination NAT. The function replaces the network-layer address and transport-layer address information in the packet with the information in the supplied tuple. The function `unique_tuple()` provides the tuple that is passed to `manip_pkt()`. The purpose of the function is to determine an available protocol-specific identifier on the external network. In the case of UDP the function `nf_nat_proto_unique_tuple()` is used to provide an available 16 bit port number. The function can be used by all protocols that use 16 bit port numbers and returns either a randomly selected and available port, or an available port from a specified range. The random port number is generated by inputting source address, destination address, destination port, and a random number into the MD5 algorithm. If a static port range has been specified, the port number is not selected randomly but from the beginning of the specified range. The `unique_tuple()` function updates the offset of the range each time it is called, and thereby ensures that returned port numbers increase monotonically within the range. The address range is provided by the `nlattr_to_range()` function. The `in_range()` function determines if a packet belongs to a group of packets that should be processed by the NAT module. If the address range is exhausted the NAT modules will begin to drop packets. The UDP NAT module utilizes the function `nf_nat_proto_in_range()`, which checks if the port number of a packet is within a range that should be processed by the NAT module. The function ensures that if a small range of port numbers is used by the NAT module, then the more complex NAT functions are only called if the packet might actually have been altered by the NAT module. If port numbers are chosen randomly the function will almost always return.

The main NAT function is `nf_nat_fn()` is called by the following hooks: `NF_INET_PRE_ROUTING`, `NF_INET_POST_ROUTING`, `NF_INET_LOCAL_OUT`, and `NF_INET_LOCAL_IN` hooks. The NAT module is called at all points in the network stack where packets enter or leave the host. The hooks are registered with a priority such that the CT module is always called before the NAT module, and the packet filtering module is always called after the NAT module. This is necessary because the NAT module depends on the states generated by the CT module.

The `nf_nat_fn()` function starts by obtaining a CT state for the packet being processed. If a CT state is not found it means that the CT module was unable to track the packet and thus it cannot be translated by NAT either. If a CT state is found and the state is `IP_CT_NEW`, the NAT rule for the packet is obtained. If no NAT rule is found, the function returns `NF_ACCEPT` without altering the packet. If a NAT rule for the packet exists, the function `nf_nat_packet()` is called. This function calls `manip_pkt()` which in turns calls the protocol-specific function by the same name as defined by a `struct nf_nat_protocol`. If the `manip_pkt()` fails to alter the packet according to the NAT rule, the packet is dropped. The return value of the protocol-specific `manip_pkt()` determines the final verdict on the packet by the NAT module.

The manipulation of the tuples generated by the CT module is performed by the function `nf_nat_setup_info()`. This function is called when a packet belonging to a new connection is sent or received. The setup function calls the `get_unique_tuple()` function which in turn calls the protocol-specific function defined in the `struct nf_nat_protocol` of the protocol. In the case of UDP, the function obtains an external IP address and port number. The tuples in the CT state are then updated with the new external IP address and port number. Due to the changes in the tuples, the hash value of the tuples will no longer point to the correct entry in the CT module's hash table. Therefore the hash value is recalculated and the CT state is moved accordingly.

5. POTENTIAL VULNERABILITIES

The design and implementation of the connection tracking and NAT modules contains several features that can make a device running NAT vulnerable to denial-of-service attacks if not configured correctly. Hash tables are inherently vulnerable to hash collision attacks. If an attacker sends a large number of packets with different source ports, the hash of

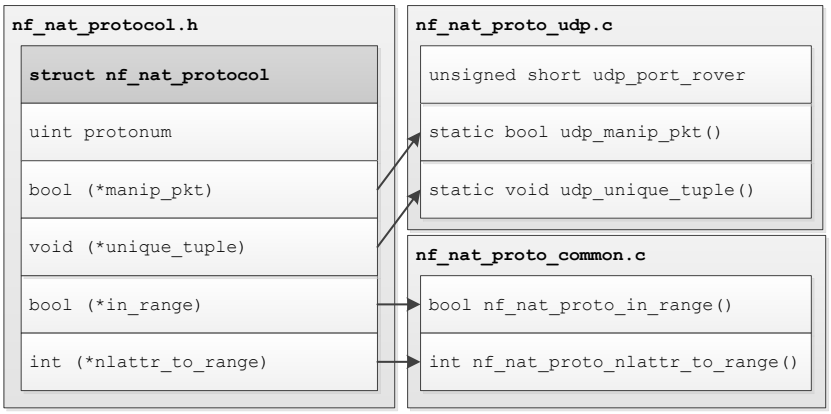

Figure 7: `struct nf_nat_protocol` and its initializations for the UDP protocol.

the tuples used by the CT module will collide depending on the size of the hash table. In Linux the default number of hash buckets depends on the amount of RAM available. For instance, a system with 4 GB of RAM has 16384 hash buckets and supports a maximum of 65536 simultaneous connections. If the maximum number of connections is reached the hash table will contain an average of four entries per hash bucket. Such a system will not become unstable if flooded by single source, but routers typically do not have 4 GB of RAM.

Another issue with NAT, is that packets are dropped if the external address space is exhausted. Flooding a gateway with packets from different source port numbers, will force NAT to allocate an external address to each flow. Because it only takes a single packet to reserve an external address, the external address space can be depleted very quickly. This problem is tied to the length of protocol timeout values. The default timeout value for unidirectional UDP traffic is 30 seconds and it is possible to send 65535 packets with all of the available source port numbers within this timeout period. A shorter timeout period results in external addresses being freed more quickly and thus denial of service is less likely. The timeout should not be set lower than the typical RTT of a connection in order to avoid replies to legitimate traffic from being dropped.

A third issues with NAT, is that connection tracking may requires transport-layer information or even application-layer information. In order to inspect data at these layers, fragmented IP packets must be defragmented and this is problematic. The defragmentation process requires the NAT device to buffer incoming packets until the whole IP packet can be assembled. The buffering consumer system resources and depending on the number of fragmented packets, this may be a problem. Because of this, some NAT devices refuse to process fragmented IP packets. Like the hash-collision attack, it may be possible to increase memory usage in a NAT device by sending many fragmented packets.

Obvious solutions to hash-collision and address exhaustion attacks are to allocate more memory to the hash table and reduce timeout values for connections. Another solution could be to dynamically adjust hash table sizes and timeout values to mitigate attacks. Dynamic adjustment of timeout values would provide better service when a device is not heavily loaded. With respect to the fragmentation issue, the simplest solution is to drop fragmented IP packets.

6. CONCLUSIONS

Netfilter is a complex and important part of the Linux kernel. This paper has given a detailed overview of the core functions and data structures used by the connection tracking and NAT modules. Some issues in connection tracking and NAT in general were also discussed. Hopefully this paper will serve as a future reference for developers who wish to contribute to the development of Linux and Netfilter.

7. REFERENCES

[1] J. Engelhardt and N. Bouliane. Writing netfilter modules, July 2012. `http://inai.de/documents/Netfilter_Modules.pdf`.

[2] IANA. Address family numbers. October 2012. `http://www.iana.org/assignments/address-family-numbers/address-family-numbers.xml`.

[3] IANA. Assigned internet protocol numbers. October 2012. `http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml`.

[4] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler. The linux networking architecture: Design and implementation of network protocols in the linux kernel. August 2004. ISBN 978-0131777200.