Netfilter Connection Tracking and NAT Implementation

Magnus Boye
Aalto University
Otakaari 5
Espoo, Finland
magnus.boye@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 Due to the constant development, existing documentation quickly becomes outdated, although the general architecture of the kernel rarely changes radically.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. Understanding the architecture and implementation of these modules is necessary in order to modify or extend this part of 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 state-full system that keeps track of incoming and outgoing flows from a network. NAT ensures that outgoing flows are mapped to a unique combination of IP address and transport-layer identifier on the external network. 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.

This paper is structured as follows. Section 2 briefly describes prior work related to Netfilter's connection tracking and NAT modules. Section 3 gives a short introduction to the Netfilter hooks used Netfilter modules. Section 4 gives an overview of the connection tracking module, its key data structure and functions. Section 5 gives an overview of the NAT module and how it relates to the connection tracking module. Section 6 describes potential vulnerabilities in the implementations of connection tracking and NAT.

2. RELATED WORK

The book by K. Wehrle et al.[2] 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.

3. 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 1 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 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 Netfilter hooks, a priority of the functions is given. This priority determines in which order 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 this is where connection tracking and NAT is implemented.

The connection tracking and NAT modules support several transport-layer protocols. In order to give a more concise overview of the modules, it was decided to focus on the generic aspects of the modules, and UDP over IPv4.

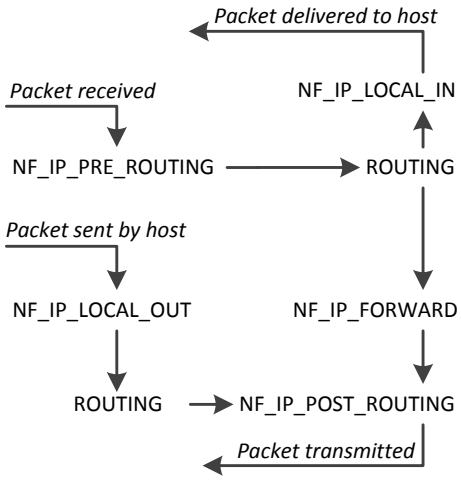

Figure 1: Netfilter IPv4 hook traversal.

4. 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.

4.1 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 2 shows a simplified representation of `struct nf_conntrack_tuple`. The data structure use `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 what properties of the different protocol headers are used to uniquely identify a packet flow. For instance, TCP and UDP use port numbers while ICMP uses ICMP type and code.

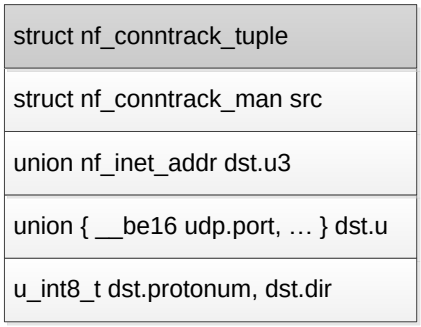

Figure 2: `struct nf_conntrack_tuple`

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 conforms 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 3 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 3 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[3][4], although some of them overlap.

As the transport-layer protocol may be connection-oriented, the `nf_conntrack_l4proto` structure contains additional function pointers that can be called depending on the state of a connection. 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. The `packet()` function is called for all packets which are deemed trackable by `error()`. Finally, the `destroy()` function is called when a connection is destroyed.

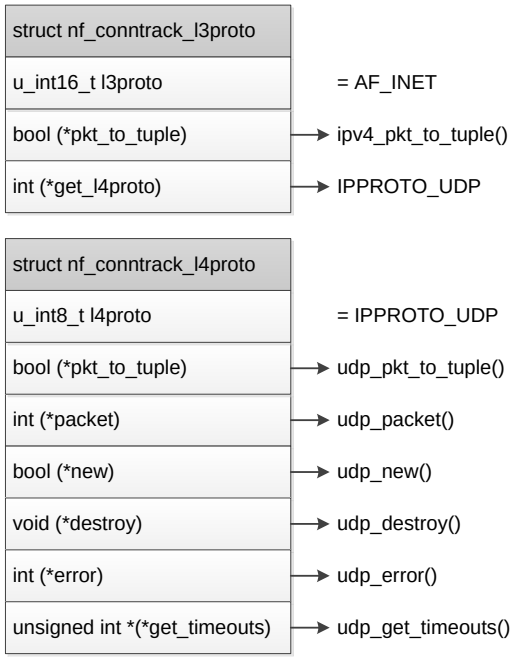

Figure 3: `struct nf_conntrack_l3proto` **and** `struct nf_conntrack_l4proto`

4.2 Hashing

The CT module is optimized for performance and therefore stores the state of active connections in a hash table. The function `hash_conntrack_raw()` returns a 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 a pointer to a linked list of CT state associated with the tuple. The linked list is used to handle hash collisions.

4.3 Connections

Netfilter uses the term *connection* even for packet flows in connectionless protocols. For the sake of clarity the same term is adopted in this paper. A tracked connection is represented by a `struct nf_conn` which is shown in Figure 4. 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 are typically timers that handle protocol timeouts and connection expiration. 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 4.3.

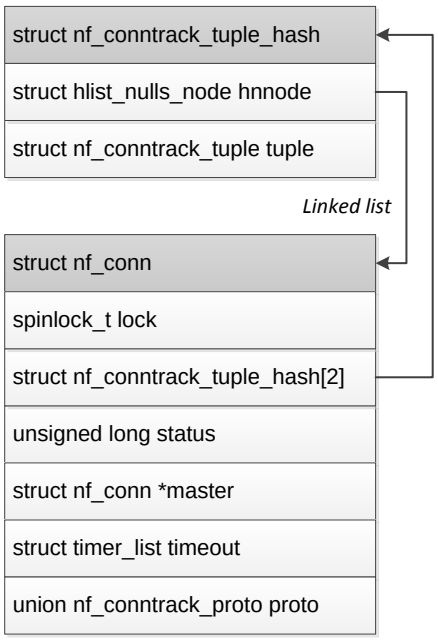

Figure 4: `struct nf_conntrack_tuple_hash` and `struct nf_conn`

4.4 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

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 1: Connection status values.

steps of the `nf_conntrack_in()` function is to determine the 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 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 another CT state and sets the `master` field of the `nf_conn` structure accordingly. The connection-less nature of UDP never results in any connection expectations. Other protocols such as FTP may expect traffic depending on certain FTP commands. Finally 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 other modules. 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 for 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. The reason for this behavior is that the CT module attempts to limit the number of half-open connections. The shorter timeout for unidirectional connections does not limit connectivity, but requires more frequent packet transmissions to prevent the connection 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 the `packet()` function, which in the case of UDP is always `NF_ACCEPT`.

5. NETWORK ADDRESS TRANSLATION

Like the CT module, the NAT module consists of a core NAT module with functions that call functions in protocol-specific modules. 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 5. 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 are 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 modules is called at all points in the network stack where packets are entering or leaving 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 in order to obtain an external IP address and port number in the case of UDP. 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.

The alteration of transport-layer addressing information proved easier to understand than the alteration of network-layer addressing information. A third Netfilter modules appears to be performing the actual swapping of IP addresses. The Masquerade module of Netfilter ensures that packets sent on a specific interface always have a source IP address matching that of the transmission interface. After modifying the source address of outgoing packets, the module creates a NAT rule such that the destination address of incoming packets will be swapped to the original address for packets in the reply direction. The inner workings of the Masquerade module and network-layer NAT is an area that needs further documentation.

6. 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 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 buck-

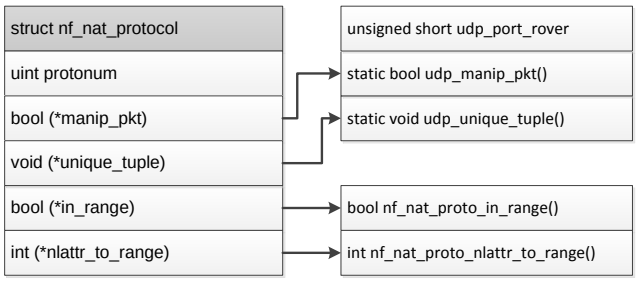

Figure 5: `struct nf_nat_protocol` and its initializations for the UDP protocol.

ets 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 obvious solutions to hash-collision and address exhaustion attacks are to allocate more memory to the hash table and reducing 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.

7. 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. Hopefully this will serve as a future reference for developers who wish to contribute to the development of Linux, modify, or extend Netfilter.

8. REFERENCES

[1] J. E. et al. Writing netfilter modules, July 2012. `http://inai.de/documents/Netfilter_Modules.pdf`.

[2] K. W. et al. The linux networking architecture: Design and implementation of network protocols in the linux kernel. August 2004.

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

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