**Introduction**

[Firewalls](https://www.digitalocean.com/community/tutorials/what-is-a-firewall-and-how-does-it-work) are an important tool that can be configured to protect your servers and infrastructure. In the Linux ecosystem, iptables is a widely used firewall tool that interfaces with the kernel’s netfilter packet filtering framework. For users and administrators who don’t understand the architecture of these systems, creating reliable firewall policies can be daunting, not only due to challenging syntax, but also because of number of interrelated parts present in the framework.

In this guide, we will dive into the iptables architecture with the aim of making it more comprehensible for users who need to build their own firewall policies. We will discuss how iptables interacts with netfilter and how the various components fit together to provide a comprehensive filtering and mangling system.

**What Are IPTables and Netfilter?**

The basic firewall software most commonly used in Linux is called iptables. The iptables firewall works by interacting with the packet filtering hooks in the Linux kernel’s networking stack. These kernel hooks are known as the netfilter framework.

Every packet that enters networking system (incoming or outgoing) will trigger these hooks as it progresses through the stack, allowing programs that register with these hooks to interact with the traffic at key points. The kernel modules associated with iptables register at these hooks in order to ensure that the traffic conforms to the conditions laid out by the firewall rules.

**Netfilter Hooks**

There are five netfilter hooks that programs can register with. As packets progress through the stack, they will trigger the kernel modules that have registered with these hooks. The hooks that a packet will trigger depends on whether the packet is incoming or outgoing, the packet’s destination, and whether the packet was dropped or rejected at a previous point.

The following hooks represent various well-defined points in the networking stack:

* NF\_IP\_PRE\_ROUTING: This hook will be triggered by any incoming traffic very soon after entering the network stack. This hook is processed before any routing decisions have been made regarding where to send the packet.
* NF\_IP\_LOCAL\_IN: This hook is triggered after an incoming packet has been routed if the packet is destined for the local system.
* NF\_IP\_FORWARD: This hook is triggered after an incoming packet has been routed if the packet is to be forwarded to another host.
* NF\_IP\_LOCAL\_OUT: This hook is triggered by any locally created outbound traffic as soon it hits the network stack.
* NF\_IP\_POST\_ROUTING: This hook is triggered by any outgoing or forwarded traffic after routing has taken place and just before being put out on the wire.

Kernel modules that wish to register at these hooks must provide a priority number to help determine the order in which they will be called when the hook is triggered. This provides the means for multiple modules (or multiple instances of the same module) to be connected to each of the hooks with deterministic ordering. Each module will be called in turn and will return a decision to the netfilter framework after processing that indicates what should be done with the packet.

**IPTables Tables and Chains**

The iptables firewall uses tables to organize its rules. These tables classify rules according to the type of decisions they are used to make. For instance, if a rule deals with network address translation, it will be put into the nat table. If the rule is used to decide whether to allow the packet to continue to its destination, it would probably be added to the filter table.

Within each iptables table, rules are further organized within separate “chains”. While tables are defined by the general aim of the rules they hold, the built-in chains represent the netfilter hooks which trigger them. Chains basically determine *when* rules will be evaluated.

As you can see, the names of the built-in chains mirror the names of the netfilter hooks they are associated with:

* PREROUTING: Triggered by the NF\_IP\_PRE\_ROUTING hook.
* INPUT: Triggered by the NF\_IP\_LOCAL\_IN hook.
* FORWARD: Triggered by the NF\_IP\_FORWARD hook.
* OUTPUT: Triggered by the NF\_IP\_LOCAL\_OUT hook.
* POSTROUTING: Triggered by the NF\_IP\_POST\_ROUTING hook.

Chains allow the administrator to control where in a packet’s delivery path a rule will be evaluated. Since each table has multiple chains, a table’s influence can be exerted at multiple points in processing. Because certain types of decisions only make sense at certain points in the network stack, every table will not have a chain registered with each kernel hook.

There are only five netfilter kernel hooks, so chains from multiple tables are registered at each of the hooks. For instance, three tables have PREROUTING chains. When these chains register at the associated NF\_IP\_PRE\_ROUTING hook, they specify a priority that dictates what order each table’s PREROUTING chain is called. Each of the rules inside the highest priority PREROUTING chain is evaluated sequentially before moving onto the next PREROUTING chain. We will take a look at the specific order of each chain in a moment.

**Which Tables are Available?**

Let’s step back for a moment and take a look at the different tables that iptables provides. These represent distinct sets of rules, organized by area of concern, for evaluating packets.

**The Filter Table**

The filter table is one of the most widely used tables in iptables. The filter table is used to make decisions about whether to let a packet continue to its intended destination or to deny its request. In firewall parlance, this is known as “filtering” packets. This table provides the bulk of functionality that people think of when discussing firewalls.

**The NAT Table**

The nat table is used to implement network address translation rules. As packets enter the network stack, rules in this table will determine whether and how to modify the packet’s source or destination addresses in order to impact the way that the packet and any response traffic are routed. This is often used to route packets to networks when direct access is not possible.

**The Mangle Table**

The mangle table is used to alter the IP headers of the packet in various ways. For instance, you can adjust the TTL (Time to Live) value of a packet, either lengthening or shortening the number of valid network hops the packet can sustain. Other IP headers can be altered in similar ways.

This table can also place an internal kernel “mark” on the packet for further processing in other tables and by other networking tools. This mark does not touch the actual packet, but adds the mark to the kernel’s representation of the packet.

**The Raw Table**

The iptables firewall is stateful, meaning that packets are evaluated in regards to their relation to previous packets. The connection tracking features built on top of the netfilter framework allow iptables to view packets as part of an ongoing connection or session instead of as a stream of discrete, unrelated packets. The connection tracking logic is usually applied very soon after the packet hits the network interface.

The raw table has a very narrowly defined function. Its only purpose is to provide a mechanism for marking packets in order to opt-out of connection tracking.

**The Security Table**

The security table is used to set internal SELinux security context marks on packets, which will affect how SELinux or other systems that can interpret SELinux security contexts handle the packets. These marks can be applied on a per-packet or per-connection basis.

**Which Chains are Implemented in Each Table?**

We have talked about tables and chains separately. Let’s go over which chains are available in each table. Implied in this discussion is a further discussion about the evaluation order of chains registered to the same hook. If three tables have PREROUTING chains, in which order are they evaluated?

The following table indicates the chains that are available within each iptables table when read from left-to-right. For instance, we can tell that the raw table has both PREROUTING and OUTPUT chains. When read from top-to-bottom, it also displays the order in which each chain is called when the associated netfilter hook is triggered.

A few things should be noted. In the representation below, the nat table has been split between DNAT operations (those that alter the destination address of a packet) and SNAT operations (those that alter the source address) in order to display their ordering more clearly. We have also include rows that represent points where routing decisions are made and where connection tracking is enabled in order to give a more holistic view of the processes taking place:

| **Tables↓/Chains→** | **PREROUTING** | **INPUT** | **FORWARD** | **OUTPUT** | **POSTROUTING** |
| --- | --- | --- | --- | --- | --- |
| (routing decision) |  |  |  | ✓ |  |
| **raw** | ✓ |  |  | ✓ |  |
| (connection tracking enabled) | ✓ |  |  | ✓ |  |
| **mangle** | ✓ | ✓ | ✓ | ✓ | ✓ |
| **nat** (DNAT) | ✓ |  |  | ✓ |  |
| (routing decision) | ✓ |  |  | ✓ |  |
| **filter** |  | ✓ | ✓ | ✓ |  |
| **security** |  | ✓ | ✓ | ✓ |  |
| **nat** (SNAT) |  | ✓ |  |  | ✓ |

As a packet triggers a netfilter hook, the associated chains will be processed as they are listed in the table above from top-to-bottom. The hooks (columns) that a packet will trigger depend on whether it is an incoming or outgoing packet, the routing decisions that are made, and whether the packet passes filtering criteria.

Certain events will cause a table’s chain to be skipped during processing. For instance, only the first packet in a connection will be evaluated against the NAT rules. Any nat decisions made for the first packet will be applied to all subsequent packets in the connection without additional evaluation. Responses to NAT'ed connections will automatically have the reverse NAT rules applied to route correctly.

**Chain Traversal Order**

Assuming that the server knows how to route a packet and that the firewall rules permit its transmission, the following flows represent the paths that will be traversed in different situations:

* **Incoming packets destined for the local system**: PREROUTING -> INPUT
* **Incoming packets destined to another host**: PREROUTING -> FORWARD -> POSTROUTING
* **Locally generated packets**: OUTPUT -> POSTROUTING

If we combine the above information with the ordering laid out in the previous table, we can see that an incoming packet destined for the local system will first be evaluated against the PREROUTING chains of the raw, mangle, and nat tables. It will then traverse the INPUT chains of the mangle, filter, security, and nat tables before finally being delivered to the local socket.

**IPTables Rules**

Rules are placed within a specific chain of a specific table. As each chain is called, the packet in question will be checked against each rule within the chain in order. Each rule has a matching component and an action component.

**Matching**

The matching portion of a rule specifies the criteria that a packet must meet in order for the associated action (or “target”) to be executed.

The matching system is very flexible and can be expanded significantly with iptables extensions available on the system. Rules can be constructed to match by protocol type, destination or source address, destination or source port, destination or source network, input or output interface, headers, or connection state among other criteria. These can be combined to create fairly complex rule sets to distinguish between different traffic.

**Targets**

A target is the action that are triggered when a packet meets the matching criteria of a rule. Targets are generally divided into two categories:

* **Terminating targets**: Terminating targets perform an action which terminates evaluation within the chain and returns control to the netfilter hook. Depending on the return value provided, the hook might drop the packet or allow the packet to continue to the next stage of processing.
* **Non-terminating targets**: Non-terminating targets perform an action and continue evaluation within the chain. Although each chain must eventually pass back a final terminating decision, any number of non-terminating targets can be executed beforehand.

The availability of each target within rules will depend on context. For instance, the table and chain type might dictate the targets available. The extensions activated in the rule and the matching clauses can also affect the availability of targets.

**Jumping to User-Defined Chains**

We should mention a special class of non-terminating target: the jump target. Jump targets are actions that result in evaluation moving to a different chain for additional processing. We’ve talked quite a bit about the built-in chains which are intimately tied to the netfilter hooks that call them. However, iptables also allows administrators to create their own chains for organizational purposes.

Rules can be placed in user-defined chains in the same way that they can be placed into built-in chains. The difference is that user-defined chains can only be reached by “jumping” to them from a rule (they are not registered with a netfilter hook themselves).

User-defined chains act as simple extensions of the chain which called them. For instance, in a user-defined chain, evaluation will pass back to the calling chain if the end of the rule list is reached or if a RETURN target is activated by a matching rule. Evaluation can also jump to additional user-defined chains.

This construct allows for greater organization and provides the framework necessary for more robust branching.

**IPTables and Connection Tracking**

We introduced the connection tracking system implemented on top of the netfilter framework when we discussed the raw table and connection state matching criteria. Connection tracking allows iptables to make decisions about packets viewed in the context of an ongoing connection. The connection tracking system provides iptables with the functionality it needs to perform “stateful” operations.

Connection tracking is applied very soon after packets enter the networking stack. The raw table chains and some basic sanity checks are the only logic that is performed on packets prior to associating the packets with a connection.

The system checks each packet against a set of existing connections. It will update the state of the connection in its store if needed and will add new connections to the system when necessary. Packets that have been marked with the NOTRACK target in one of the raw chains will bypass the connection tracking routines.

**Available States**

Connections tracked by the connection tracking system will be in one of the following states:

* NEW: When a packet arrives that is not associated with an existing connection, but is not invalid as a first packet, a new connection will be added to the system with this label. This happens for both connection-aware protocols like TCP and for connectionless protocols like UDP.
* ESTABLISHED: A connection is changed from NEW to ESTABLISHED when it receives a valid response in the opposite direction. For TCP connections, this means a SYN/ACK and for UDP and ICMP traffic, this means a response where source and destination of the original packet are switched.
* RELATED: Packets that are not part of an existing connection, but are associated with a connection already in the system are labeled RELATED. This could mean a helper connection, as is the case with FTP data transmission connections, or it could be ICMP responses to connection attempts by other protocols.
* INVALID: Packets can be marked INVALID if they are not associated with an existing connection and aren’t appropriate for opening a new connection, if they cannot be identified, or if they aren’t routable among other reasons.
* UNTRACKED: Packets can be marked as UNTRACKED if they’ve been targeted in a raw table chain to bypass tracking.
* SNAT: A virtual state set when the source address has been altered by NAT operations. This is used by the connection tracking system so that it knows to change the source addresses back in reply packets.
* DNAT: A virtual state set when the destination address has been altered by NAT operations. This is used by the connection tracking system so that it knows to change the destination address back when routing reply packets.

The states tracked in the connection tracking system allow administrators to craft rules that target specific points in a connection’s lifetime. This provides the functionality needed for more thorough and secure rules.

**Conclusion**

The netfilter packet filtering framework and the iptables firewall are the basis for most firewall solutions on Linux servers. The netfilter kernel hooks are close enough to the networking stack to provide powerful control over packets as they are processed by the system. The iptables firewall leverages these capabilities to provide a flexible, extensible method of communicating policy requirements to the kernel. By learning about how these pieces fit together, you can better utilize them to control and secure your server environments.

If you would like to know more about how to choose effective iptables policies, check out [this guide](https://www.digitalocean.com/community/tutorials/how-to-choose-an-effective-firewall-policy-to-secure-your-servers).

These guides can help you get started implementing your iptables firewall rules:

* [How To Set Up a Firewall Using Iptables on Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/how-to-set-up-a-firewall-using-iptables-on-ubuntu-14-04)
* [Iptables Essentials: Common Firewall Rules and Commands](https://www.digitalocean.com/community/tutorials/iptables-essentials-common-firewall-rules-and-commands)
* [How To Implement a Basic Firewall Template with Iptables on Ubuntu 14.04](https://www.digitalocean.com/community/tutorials/how-to-implement-a-basic-firewall-template-with-iptables-on-ubuntu-14-04)
* [How To Set Up an Iptables Firewall to Protect Traffic Between your Servers](https://www.digitalocean.com/community/tutorials/how-to-set-up-an-iptables-firewall-to-protect-traffic-between-your-servers)