NPF in NetBSD 6

S.P.Zeidler ⟨*spz@NetBSD.org*⟩
The NetBSD Foundation

Mindaugas Rasiukevicius ⟨*rmind@NetBSD.org*⟩
The NetBSD Foundation

Abstract

NPF has been released with NetBSD 6.0 as an experimental packet filter, and thus has started to see actual use. While it is going to take a few more cycles before it is fully "production ready", the exposure to users has given it a strong push to usability. Fixing small bugs and user interface intuitivity misses will help to evolve it from a theoretical well-designed framework to a practical packet filtering choice. The talk will cover distinguishing features of NPF design, give an overview of NPF's current practical capabilities, ongoing development, and will attempt to entice more people to try out NPF and give feedback.

1 Introduction

NPF is a layer 3 packet filter, supporting IPv4 and IPv6, as well as layer 4 protocols such as TCP, UDP and ICMP/IPv6-ICMP. NPF offers the traditional set of features provided by most packet filters. This includes stateful packet filtering, network address translation (NAT), tables (using a hash or tree as a container), rule procedures for easy development of NPF extensions (e.g. packet normalisation and logging), connection saving and restoring as well as other features. NPF focuses on high performance design, ability to handle a large volume of clients and using the speed of multi-core systems.

Various new features were developed since NetBSD 6.0 was released, and the upcoming 6.1 release will have considerable differences regarding their user interface and to a certain level regarding its capabilities.

2 What's special about NPF?

Inspired by the Berkeley Packet Filter (BPF), NPF uses "n-code", which is conceptually a byte-code processor, similar to machine code. Each rule is described by a sequence of low level operations, called "n-code", to perform for a packet. This design has the advantage of protocol independence, therefore support for new protocols (for example, layer 7) or custom filtering patterns can be easily added at userspace level without any modifications to the kernel itself.

NPF provides rule procedures as the main interface to use custom extensions. The syntax of the configuration file supports arbitrary procedures with their parameters, as supplied by the extensions. An extensions consists of two parts: a dynamic module (.so file) supplementing the npfctl(8) utility and a kernel module (.kmod file). Thus, kernel interfaces can be used instead of modifications to the NPF core code.

The internals of NPF are abstracted into well defined modules and follow strict interfacing principles to ease extensibility. Communication between userspace and the kernel is provided through the library libnpf, described in the npf(3) manual page. It can be conveniently used by developers who create their own extensions or third party products based on NPF. Application-level gateways (ALGs), such as support for traceroute(8), are also abstracted in separate modules.

2.1 Designed for SMP and high throughput

NPF has been designed so its data structures can use lockless methods where suitable and fine-grained locking in general. [1]

Ruleset inspection is lockless. It uses passive serialization as a protection mechanism. The reload of a ruleset is atomic with minimum impact on the active ruleset, i.e. the rule processing is not blocked during the reload. NPF rules can be nested, which is useful for grouping

[1] For the initial NPF release, some components are using read-write locks, although they could be lockless using a passive serialization interface. Since this interface was new in NetBSD 6.0, a conservative approach was taken. As of 6.1, those components have been converted to be lockless.

and chaining based on certain filtering patterns. Currently, the configuragion file syntax supports two levels of groups (having per-interface and traffic direction options), however there is no limitation in the kernel and syntax expansion is planned. As of NetBSD 6.1, dynamic NPF rules will be supported.

Efficient data structures were chosen for the connection (session) tracking mechanism: a hash table with buckets formed of red-black trees. The hash table provides distribution of locks which are protecting the trees, thus reducing lock and cacheline contention. The tree itself provides efficient lookup time in case of hash collision and, more importantly, prevents algorithmic complexity attacks on the hash table i.e. its worst case behaviour.

The session structure is relatively protocol-agnostic. The IPv4 or IPv6 addresses are used as the first set of IDs forming a key. The second set of IDs are generic. Depending on the protocol, they are filled either with port numbers in the TCP/UDP case or with ICMP IDs. It should be noted that the interface is also part of the key, as is the protocol. Therefore, a full key for a typical TCP connection would be formed from: `SRC_IP:SRC_PORT:DST_IP:DST_PORT:PROTO:IFACE`. This key is a unique identifier of a session.

Structures carrying information about NAT and/or rule procedures are associated with the sessions and follow their life-cycle.

NPF provides efficient storage for large volumes of IP addresses. They can be stored in a hash table or in a Patricia radix tree. The latter also allows to specify address ranges.

2.2 Modular design

NPF is modular, each component has its own abstracted interface. This makes writing NPF extensions easy. It also allows easy addition of filtering capabilities for layer 4 and higher. The implementer of a network protocol does not need to know anything about the internals of packet collection and disposal facilities. Services such as connection tracking are provided by a strict interface - other components may consider it as a black box.

The NPF extensions API will be fully provided with the NetBSD 6.1 release. Extensions consist of kernel modules and userland modules implemented as dynamically loadable libraries for npfctl, the NPF control utility. Extensions get configured as rule procedures and applied on the selected packets. They can take arguments in a key-value form. Extensions may rewrite packet contents (e.g. fields in the header) and influence their fate (block/pass).

There is a demo extension: the kernel part in `src/sys/net/npf/npf_ext_rndblock.c` and `src/lib/npf/ext_rndblock/npfext_rndblock.c` for npfctl. This extension simulates packet loss. The kernel part file is less than 180 lines long, the npfctl part is less than 100. Given that the copyright notice is a significant part of that, the 'administrative overhead' for a NPF extension is fairly low.

3 What can it do, at present?

The configuration file syntax is still avoiding to be Turing complete. In spite of the obvious temptation, it is planned to keep it that way. The config syntax has changed noticeably between what was released with NetBSD 6.0, and what will be in NetBSD 6.1. Further change is expected (of course, only to the better).

As mentioned, NPF is a stateful packet filter for IP (v4 and v6) and layer 4 - TCP, UDP, ICMP and ICMPv6, including filtering for ports, TCP states, ICMP types and codes are currently implemented.

Configuration example:

```
pass in family inet proto tcp \
  from $nicehost to $me port ssh
```

Rules get processed top to bottom as they are written in the config file, first through the interface specific group and then through the default group. Processing can be stopped early by using the tag `final` in a rule.

Addresses for filtering can be inferred from what is configured on an interface (own addresses), can be configured in the npf.conf configuration file, or can be fed into (and deleted from) tables defined in npf.conf using `npfctl table` commands.

NPF sidesteps the fragments issues by reassembling packets before further processing.

NPF supports various address and port translation variants. In NetBSD 6.0.x, it supports network address port translation ("masquerading"), port forwarding ("redirection") and bi-directional NAT in IPv4. There is an application-level gateway (ALG) for traceroute and ICMP translation; ALGs are provided as kernel modules.

NetBSD 6.0 configuration example:

```
# outgoing NAPT
map $ext_if dynamic 198.51.100.0/24 -> \
  $ext_if
# port forwarding
map $ext_if dynamic 198.51.100.2 port 22 \
  <- $ext_if 9022
```

Session state (including NAT state) can be dumped to file in a serialised form.

Packet normalization and logging are available since the NetBSD 6.0 release. From 6.1 onwards, they will be provided as NPF extensions. They get configured as procedures:

```
# procedure to log a packet and
# decrease its MSS to 1200 if higher
procedure "maxmss_log" {
  log: npflog0
  normalise: "max-mss" 1200
}
# in an interface group, apply the
# procedure to matching packets:
pass out family inet proto tcp flags S/SA \
  to $pmtublackholed apply "maxmss_log"
# continue to further processing
```

Rule reload builds a new configuration in the kernel, and switches from new to old atomically. NPF does not hold several configurations in the kernel to switch between. Instead, support for a temporary load with a timer and automatic rollback is planned.

4 Testing and debugging

As a good citizen of NetBSD, NPF has regression tests in the automated tests framework. These are implemented using the RUMP (Runnable Userspace Meta Programs) framework, which allows to exercise the kernel elements of NPF in userspace, without having to boot a test kernel (even in a VM). Regular tools like gdb can be used for debugging. Unit tests are implemented and available via the npftest program. Additionally, the program can load a custom configuration file and process packets from a pcap file. This allows developers to analyse NPF decisions and state tracking in userspace using a captured sample. As a result, debugging the problems experienced in other environments is easier.

Another debugging help is the npfctl debug option, which dumps the internal representation of a parsed config, as npfctl would have sent to the kernel, i.e. as disassembled n-code.

5 Meeting users

The NetBSD 6.0 configuration file syntax uses one label for a network interface, the addresses configured on that interface and the addresses configured for a specific address family on an interface; the meaning was inferred from context. This turned out to be hard to understand for humans (even if the software interpreted it just fine) and was changed for NetBSD 6.1 to explicitly require a function that takes the interface name as an argument, e.g. `inet($interface)` for all addresses configured on `$interface`.

Even before NetBSD 6.0 was shipped, the syntax expressing address and port translations was reworked to use `map`. It is bound to an interface, the crossing of which triggers the translation. By convention, the ”inside” network and optional port definition is on the left-hand side, and the ”outside” on the right side on the arrow that discerns the translation direction. This makes configurations easier to read (for the current sample of users).

Users also saw cosmetic issues that were previously missed, like printing garbage for linklocal scope for `npfctl show`.

Further improvement will be necessary in giving useful error messages for syntax problems in the configuration file. Getting feedback from users is very important to get NPF production ready, to oust the last small bugs and help identify the most desireable missing bits.

6 Whereto, my lovely?

NPF is still a moving target - there have been major changes and feature additions recently. Although the core APIs of NPF are becoming stable, they are still subject to extensions and minor changes. Therefore, when developing 3rd party extensions, it is recommended to follow source-changes@NetBSD.org since ”catching up” with the changes might be necessary.

NPF will also use BPF byte-code (first available in NetBSD 6.1), and BPF with just-in-time (JIT) compilation likely in NetBSD 7.0. In combination with the widely used pcap(3) library, it provides the ability to write sophisticated filtering criteria in an easy, expressive and very efficient way that is familiar to users.

A GSoC 2012 project produced patches for simple-case NPT66 (/48 only) and NAT64 (well-known prefix only). However, these still need to be expanded and integrated, and are therefore unlikely to make it into NetBSD 6.1 due to ”free time” constraints.

Further extensions are likely to arrive. Possible examples would be extensions for port knocking, traffic accounting and connection rate limiting.

Given interest, addition of further layer 4 protocols is likely.

Finally, NPF has design provisions to eventually enable active-passive and active-active firewall clusters; no timeline for implementing this exists yet.