NPF – the packet filter of NetBSD

Mindaugas Rasiukevicius

October 17, 2012

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. NPF offers a 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, packet normalisation and logging, connection saving and restoring and more. NPF focuses on high performance design, ability to handle large volume of clients and use of the speed of multi-core systems. Our main objective for the preview release is to stabilise initial set of features and have a robust code base for further development.

This paper covers NPF version distributed by the NetBSD 6.0 release.

2 Brief notes on design

Inspired by the Berkeley Packet Filter (BPF), NPF uses ”n-code”, which is conceptually a byte-code processor, similar to the 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. BPF just-in-time (JIT) compilation itself is planned for the future release.

NPF provides rule procedures as the main interface to implement custom extensions. 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). Kernel interfaces are available for use and avoid modifications to the NPF core code.

The internals of NPF are abstracted into well defined modules and follow strict interfacing principles to ease the 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 the 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.

3 Configuration

The first step is configuring general networking settings in the system, for example assigning the addresses and bringing up the interfaces. NetBSD beginners can consult rc(8), rc.conf(5), ifconfig.if(5) and other manual pages. The second step is to create NPF's configuration file (by default, /etc/npf.conf). We will give an overview with some simple and practical examples. A detailed description of the syntax and options is provided in the npf.conf(5) manual page. The following is a simplistic configuration, which contains two groups for two network interfaces and a default group:

```
srv# cat /etc/npf.conf
$ext_if = "wm0"
$int_if = "wm1"

group (name "external", interface $ext_if) {
        pass all
}

group (name "internal", interface $int_if) {
        pass all
}

group (default) {
        pass final on lo0 all
        block all
}
```

It will be explained in detail below.

3.1 Control

NPF can be controlled through the npfctl(8) utility or NetBSD's rc.d system. The latter is used during system startup, but essentially it is a convenience wrapper. The following is an example for starting NPF and loading the configuration through the rc.d script:

```
srv# echo 'npf=YES' >> /etc/rc.conf
srv# /etc/rc.d/npf reload
Reloading NPF ruleset.
srv# /etc/rc.d/npf start
Enabling NPF.

srv# npfctl
usage:  npfctl [ start | stop | reload | flush | show | stats ]
        npfctl ( sess-save | sess-load )
        npfctl table <tid> [ flush ]
        npfctl table <tid> { add | rem | test } <address/mask>
```

Any modifications of npf.conf require reloading of the ruleset by performing a 'reload' command in order to make the changes active. One difference from other packet filters is the behaviour of the 'start' and 'stop' commands. These commands do not actually change (i.e. load or unload) the active configuration. Running 'start' will only enable the passing of packets through NPF, while 'stop' will disable such passing. Therefore, configuration should first be activated using 'reload' command and then filtering enabled with 'start'. Similarly, clearing of the active configuration is done by performing the 'stop' and 'flush' commands. Such behaviour allows users to efficiently disable and enable filtering without actually changing the active configuration, which might be unnecessary and expensive.

3.2 Definitions

Definitions are general purpose keywords which can be used in the ruleset to make it more flexible and easier to manage. Most commonly, definitions are used to define one of the following: IP addresses, networks, ports or interfaces. Definitions can contain multiple elements.

In the example above, network interfaces are defined using the $ext_if and $int_if variables (note that the dollar sign ('$') indicates a variable), which can be used further in the configuration file.

3.3 Groups

Having one huge ruleset for all interfaces or directions might be inefficient; therefore, NPF requires that all rules be defined within groups. Groups can be thought of as higher level rules which can contain subrules. The main properties of a group are its interface and traffic direction. Packets matching group criteria are passed to the ruleset of that group. If a packet does not match any group, it is passed to the default group. The default group must always be defined.

In the given example, packets passing *wm0* network interface will first be inspected by the rules in the group named "external" and if none matches, the default group will be inspected. Accordingly, if the packet is passing *wm1*, group "internal" will be inspected first, etc. If the packet is neither on *wm0* nor *wm1*, then the default group would be inspected first.

3.4 Rules

Rules, which are the main part of the NPF configuration, describe the criteria used to inspect and make decisions about packets. Currently, NPF supports filtering on the following criteria: interface, traffic direction, protocol, IP address or network, TCP/UDP port or range, TCP flags and ICMP type/code. Supported actions are blocking or passing the packet.

Each rule has a priority, which is set according to its order in the ruleset. Rules defined first are accordingly inspected first. All rules in the group are inspected sequentially and the last matching one dictates the action to be taken. Rules, however, may be explicitly marked as final. In such cases, processing stops after encountering the first matching rule marked as final. If there is no matching rule in the custom group, then as described previously, rules in the default group will be inspected.

In the example, both interfaces have a ”pass all” rule, which permits any incoming and outgoing packets on these interfaces.

3.5 Tables

A common problem is the addition or removal of many IP addresses for a particular rule or rules. Reloading the entire configuration is a relatively expensive operation and is not suitable for a stream of constant changes. It is also inefficient to have many different rules with the same logic just for different IP addresses. Therefore, NPF tables are provided as a high performance container to solve this problem.

NPF tables are containers designed for large IP sets and frequent updates without reloading the entire ruleset. They are managed separately, without reloading of the active configuration. It can either be done dynamically or table contents can be loaded from a separate file, which is useful for large static tables.

There are two supported NPF table types: ”tree” and ”hash”. The underlying data structure, accordingly, is either a PATRICIA radix tree or a hash table. These data structures allow NPF to perform efficient lookups. Tree tables perform IP prefix matching, therefore both single addresses and address ranges may be added into the table. Hash tables, in contrast, can only store single IP addresses.

The following fragment is an example using two tables:

```
table <1> type hash file "/etc/npf_blacklist"
table <2> type tree dynamic

group (name "external", interface $ext_if) {
        block in final from <1>
        pass stateful out final from <2>
}
```

The static table identified with number 1 is loaded from a file (in this case, located at /etc/npf_blacklist). The dynamic table is initially empty and has to be filled once the configuration is loaded. Tables can be filled and controlled using the npfctl(8) utility. Examples to flush a table, add an entry and remove an entry from the table:

```
svr# npfctl table 1 flush
svr# npfctl table 2 add 10.0.1.0/24
svr# npfctl table 2 rem 10.0.2.1
```

A public ioctl(2) interface for applications to manage the NPF tables is also provided.

4 Stateful filtering

TCP is a connection-oriented protocol, which means that network stacks have a state structure for each connection. The state is updated during the session. A specific session is determined by the source and destination IP addresses, port numbers and the direction of the initial packet. Additionally, TCP is responsible for reliable transmission, which is achieved using TCP sequence and window numbers. Validating the data of each packet according to the data in the state structure, as well as updating the state structure, is called TCP state tracking. Since packet filters are the middle points between the hosts (i.e. senders and receivers) they have to perform their own TCP state tracking for each session in order to reliably distinguish different TCP connections and perform connection-based filtering. Heuristic algorithms are used to handle out-of-order packets, packet losses and prevent connections from malicious packet injections. Using the conceptually same technique, limited tracking of message-based protocols, mainly UDP and ICMP, can also be done. Packet filters which have the described functionality are called *stateful* packet filters.

NPF is a stateful packet filter capable of tracking TCP connections, as well as performing limited UDP and ICMP tracking. Stateful filtering is enabled using the ”stateful” keyword. In such cases, as described in the previous paragraph, a state (a session) is created and any further packets of the connection are tracked. Packets in the backwards stream, after having been confirmed to belong to the same connection, are passed without ruleset inspection. Example configuration fragment with stateful rules:

```
group (name "external", interface $ext_if) {
        block all
        pass stateful in final proto tcp flags S/SA to $ext_if port ssh
}
```

In this example, all incoming and outgoing traffic on the $ext_if interface will be blocked, with the exception of incoming SSH traffic (with the destination being an IP address of this interface) and the implicitly passed backwards stream (outgoing reply packets) of these SSH connections. Since initial TCP packets opening a connection are SYN packets, such rules often have additional TCP filter criterion. The expression *flags S/SA* extracts SYN and ACK flags and checks that SYN is set and ACK is not.

5 Network Address Translation

NPF supports various forms of network address translation (NAT). Currently, only dynamic NAT is supported, which includes traditional NAT (known as NAPT or masquerading), bi-directional NAT and port forwarding. NAT64 (protocol translation) and static NAT, including NPTv6 (prefix translation) are planned for future releases of NPF.

It should be remembered that dynamic NAT, as a concept, relies on stateful filtering, therefore it is performing it implicitly. The following is an example configuration fragment of a traditional NAPT setup:

```
map $ext_if dynamic $localnet -> $ext_if

group (name "external", interface $ext_if) {
        pass stateful out final proto tcp flags S/SA from $localnet
}
```

The first line enables traditional NAPT (keyword *map*) on the interface specified by $ext_if for all packets from the network defined in $localnet to any other network (0.0.0.0/0), where the address to translate to is the (only) one on the interface $ext_if (it may be specified directly as well, and has to be specified directly if the interface has more than one IPv4 address).

The arrow indicates the translation type, which can be one of the following:

- `->` – for outbound NAT (also known as source NAT).
- `<-` – for inbound NAT (destination NAT).
- `<->` – for bi-directional NAT.

The rule *pass ...* permits all outgoing packets from the specified network. It additionally has stateful tracking enabled with the keyword *stateful*. Therefore, any incoming packets belonging to the connections which were created by initial outgoing packets will be implicitly passed.

The following two lines are example fragments of bi-directional NAT and port 8080 forwarding to a local IP address, port 80:

```
map $ext_alt_if dynamic $local_host_1 <-> $ext_alt_if
map $ext_if dynamic $local_host_2 port 80 <- $ext_if port 8080
```

Processing order in NPF is as follows: state inspection → rule inspection (if no state) → NAT → rule procedure. We will discuss rule procedures in a further section.

6 Application Level Gateways

Certain application layer protocols are not compatible with NAT and require translation outside layer 3 and 4. Such translation is performed by the packet filter extensions called application level gateways (ALGs). Some common cases are: traceroute and FTP applications.

Support for traceroute (both ICMP and UDP cases) is built-in, unless NPF is used from kernel modules. In that case, it is enough to load the ICMP ALG kernel module:

```
modload npf_alg_icmp
```

7 Rule procedures

Rule procedures are a key interface in NPF, which is designed to perform custom actions on packets. Users can implement their own specific functionality as a kernel module extending NPF. The configuration file is flexible to accept calls to such procedures with variable arguments. Apart from syntax validation, the npfctl(8) utility has to perform extra checks while loading the configuration. It checks whether the custom procedure is registered in the kernel and whether the arguments of the procedure are valid (e.g. that the passed values are permitted). There are built-in rule procedures provided by NPF, e.g. packet logging and traffic normalisation.

The following is an example of two rule procedure definitions – one for logging and another one for normalisation:

```
procedure "log" {
        log: npflog0
}

procedure "norm" {
        normalise: "random-id", "min-ttl" 512, "max-mss" 1432
}
```

Traffic normalisation has a set of different mechanisms. In the example above, the normalisation procedure has arguments which apply the following mechanisms: IPv4 ID randomisation, minimum TTL enforcement and TCP MSS ”clamping”.

To execute the procedure for a certain rule, use the *apply* keyword:

```
group (name "external", interface $ext_if) {
        block in final from <1> apply "log"
}
```

In the case of stateful inspection (when a rule contains the *stateful* keyword), the rule procedure will be associated with the state i.e. the connection. Therefore, a rule procedure would be applied not only for the first packets which match the rule, but also for all subsequent packets belonging to the connection. It should be noted that a rule procedure is associated with the connections for their entire life cycle (until all associated connections close) i.e. a rule procedure may stay active even if it was removed from the configuration.

8 Troubleshooting

NPF provides information about the current active configuration using the npfctl(8) command ”show”. It should be noted that the output produced by this command may not conform to the syntax of the npf.conf file. It is also not recommended to use this command for high frequency requests, as it is not designed to scale.

NPF also provides general statistics via the command ”stats”. These and the packet logging interface can be used for troubleshooting. Additionally, there are debugging facilities, such as the ”debug” command and the npftest utility, which are mainly targeted at developers.