1 netsniff-ng

netsniff-ng — the packet sniffing beast

Synopsis

> *netsniff-ng* [-d netdev] [-p pcap-file] [-r pcap-file] [-i pcap-file]
> [-f bpf-file] [-t pkt-type] [-b cpu-range] [-B cpu-range]
> [-S ring-size] [-e regex] [-IMHQnsqlxCXNvh]

DESCRIPTION

netsniff-ng is is a free (GPL), performant Linux network sniffer for packet inspection.

The gain of performance is reached by *zero-copy* mechanisms, so that the kernel does not need to copy packets from kernelspace to userspace.

For this purpose netsniff-ng is libpcap independent, but nevertheless supports the pcap file format for capturing, replaying and performing offline-analysis of pcap dumps. Furthermore we are focussing on building a robust, clean and secure analyzer and utilities that complete netsniff-ng as a support for penetration testing.

netsniff-ng can be used for protocol analysis, reverse engineering and network debugging.

OPTIONS

-d <netdev> , --dev <netdev>
<netdev> defines the packet capturing device. This can for instance be a typical device like eth0 or wlan0. Running netsniff-ng without a given device parameter, it looks for up and running networking devices and selects the first device that has been found.

-p <pcap-file> , --dump <pcap-file>
netsniff-ng stores the captured packets into the given <pcap-file>. It understands the PCAP specification, so that dumps can be read or postprocessed with other tools, too. Usually, this option should be combined with --silent and --bind-cpu to win some performance.

-r <pcap-file> , --replay <pcap-file>
The given <pcap-file> will be replayed via a memory mapped kernelspace TX_RING. A BPF filter may be combined to only replay parts of the PCAP formatted file.

-i <pcap-file> , --read <pcap-file>
<pcap-file> will be read in and printed to the console in order to perform an offline analysis. Same here: a BPF may be combined to only show relevant parts of the PCAP formatted file. Next to this, packet printing that are enabled on the normal mode are supported, too.

-f <bpf-file> , --filter <bpf-file>
Attaches a Berkeley Packet Filter to the socket in order to pre-filter traffic within the kernel. Example files are given within /etc/netsniff-ng/rules/. The section *BERKELEY PACKET FILTER* describes how to write filter files.

-t <pkt-type> , --type <pkt-type>
A <pkt-type> specification allows to post-filter packets within userspace context (therefore slower than BPF). The following types are supported: host - only show incoming packets to our host, broadcast - only show Broadcast packets, multicast - only show Multicast packets, others - only show packets from other hosts (promiscuous mode), outgoing - only show outgoing packets from our host

-b <cpu-range> , --bind-cpu <cpu-range>
Force system scheduler to schedule netsniff-ng only on specific CPUs. Parameters may be *0* for using only CPU0, *0,1* for using CPU0 and CPU1 or even *0-4* for using a whole CPU range. If you have a customized init process that leaves out a special CPU you could bind netsniff-ng on that free CPU for maximal performance. On the other hand, you can avoid scheduling netsniff-ng on CPUs which are reserved for other critical tasks. This can also be combined with taskset(1) in order to reschedule other processes on other CPUs to let netsniff-ng run on its own CPU.

-B <cpu-range> , --unbind-cpu <cpu-range>
Force system scheduler to not schedule netsniff-ng on specific CPUs. The parameter syntax is equivalent to -b and also the semantics are inverted to -b.

-S <ring-size> , --ring-size <size>
This manually sets the desired ring size for RX_RING or TX_RING. You should only use this option, if you know what you are doing, because choosing a ring size which is too large for your system, the kernel does neither warn you nor throws an error. It simply kills other processes to grab their space. The parameter can be defined in KB, MB or GB as *10MB* for a 10 Megabyte ring size.

-I , --info
Shows information about available networking devices.

-M , --no-promisc
Forbids the NIC to enter the promiscuous mode. The promiscuous mode is activated by default. It is a configuration of a network card that makes the card pass all traffic it receives to the central processing unit rather than just frames addressed to it. Well, do not ask yourself why you cannot see traffic by others within a switched network. Unlike old hubs, switches are some kind of intelligent devices with internal ARP tables for each port in order to reduce traffic load and prevent sniffing other connections. If you really intent to sniff others traffic, go read about ARP cache poisoning / MITM.

-H , --prio-norm
This option prevents to automatically high priorize itself. Normally, netsniff-ng will be scheduled with high priority thus it use the full CPU timeslice.

-Q , --notouch-irq
If netsniff-ng will be bound to a single CPU, say CPU0, then it automatically rebinds the NIC interrupt affinity to that CPU, too for a better performance. This feature is intended to be enabled on non-wireless interfaces. notouch-irq forbids netsniff-ng to move the IRQ affinity.

-n , --non-block
Lets netsniff-ng run in non-blocking mode. Generally, you won't need this feature very often unless there is some interest in performance behaviour analysis. This will bypass the ring polling mechanism thus CPU load will most certainly rise to 100 percent.

-s , --silent
Does not print packets to the terminal.

-q , --less
Prints one-liner information summary per packet.

-l , --payload
Shows only the payload information of the packet.

-x , --payload-hex
Shows only the payload information of the packet in hexadecimal format.

-C , --c-style
Instead of printing packet in usual hex format, a copy-and-paste C like format will be printed to the terminal.

-X , --all-hex
Shows not only the payload in hexadecimal format, but the whole packet.

-N , --no-payload
Shows only the packet header, not the payload.

-e <regex> , --regex <expr>
Regular expression printing is useful for grepping ASCII text out of packets, say certain HTML code for instance. Beware, that this has a impact on performance. Regular expressions that comply with the POSIX extended regular expression format are allowed.

-v , --version
Shows version number and exits.

-h , --help
Shows help and exits.

BERKELEY PACKET FILTER

The Berkeley Packet Filter or BSD Packet Filter was first introduced in 1993 by Steven McCanne and Van Jacobson at the USENIX. Its purpose is to filter packets within the kernel as early as possible, so that only the relevant packets will be brought to the user-level process.

The Linux kernel has adapted this feature, which nowadays is available in PF_PACKET. BPF therefore uses a register-based filter-machine that is efficient on todays RISCs. Since most applications of a packet filter reject far more packets than they accept and, thus, good performance of the packet filter is critical to good overall performance [1]. This should also be kept in mind during development of filters.

If you don't want to write your own filters, we currently ship some examples within /etc/netsniff-ng/rules/ that can be used with netsniff-ngs -f option. Furthermore *tcpdump -dd* provides filter creation that netsniff-ng can read, but be warned - most certainly you might need to edit the return value, which defines the packet snaplen for your needs. Read section NOTES for more information about this. In future versions netsniff-ng will also ship its own filter compiler for a simple usage.

If you are an advanced user and if you would like to have full control of what should be filtered and what not, then writing your own filter could be a suitable choice. Hence, in the following the language specification will be described in short with given examples on how to use it.

The BPF pseudo-machine consists of an accumulator, an index register (X), a scratch memory store and an implicit program counter. Operations on this machine can be categorized into (all the following refering to [1]):

1. *Load instructions:* Load instructions copy a value into the accumulator or index register. The source can be an immediate value, packet data at a fixed offset, packet data at a variable offset, the packet length, or the scratch memory store.
2. *Store instructions:* Store instructions copy either the accumulator or index register into the scratch memory store.
3. *ALU instructions:* ALU instructions perform arithmetic or logic on the accumulator using the index register or a constant as an operand.
4. *Branch instructions:* Branch instructions alter the flow of control, based on comparison test between a constant or x register and the accumulator.
5. *Return instructions:* Return instructions terminate the filter and indicated what portion of the packet to save. The packet is discarded entirely if the filter returns 0.
6. *Misc instructions:* Misc instructions comprise everything else - currently, register transfer instructions.

The instruction format is of fixed length that is defined as the following:

```
+-----------+------+------+------+
| opcode:16 | jt:8 | jf:8 | k:32 |
+-----------+------+------+------+
```

The opcode field indicates the instruction type and addressing modes. The jt and jf fields are used by the conditional jump instructions and are the offsets from the next instruction to the true and false targets. The K field is a generic field used for various purposes.

All values are 32 bit words.

The Linux kernel has adapted this within linux/filter.h:

```
struct sock_filter {     /* Filter block */
        __u16   code;    /* Actual filter code */
        __u8    jt;      /* Jump true */
        __u8    jf;      /* Jump false */
        __u32   k;       /* Generic multiuse field */
};

struct sock_fprog {                        /* Required for SO_ATTACH_FILTER. */
        unsigned short             len;    /* Number of filter blocks */
        struct sock_filter __user *filter;
};
```

The instruction set is similar to assembler syntax. There are several instruction classes which are similar to the previous categorization:

```
 LD    0x00  Copy indicated value into accumulator
 LDX   0x01  Copy indicated value into index register
 ST    0x02  Copy accumulator value into the scratch memory store
 STX   0x03  Copy index register value into the scratch memory store
 ALU   0x04  Perform arithmetic or logic operation on the accumulator
 JMP   0x05  Perform a branch instruction
 RET   0x06  Return/exit from filter program
 MISC  0x07  Data transfer between index register and accumulator
```

Next to classes, there are class-specific fields which are usually combined with bitwise OR:

LD/LDX specific fields:

```
 Size:
  W    0x00  Unsigned Word (32 Bit)
  H    0x08  Unsigned Halfword (16 Bit)
  B    0x10  Unsigned Byte

 Mode:
  IMM  0x00  Literal value stored in K
  ABS  0x20  Byte, halfword or word at offset K in the packet
  IND  0x40  Byte, halfword or word at offset X + K in the packet
  MEM  0x60  Word at offset K in the scratch memory store
  LEN  0x80  Length of the packet
  MSH  0xa0  4*([K]&0xf): four times the value of the low four bits
             of the byte at offset K in the packet
```

ALU/JMP operations perform the indicated operation using the accumulator and operand, and store the result back into the accumulator. Division by zero terminates the filter.

ALU/JMP specific fields:

```
 Operation:
  ADD  0x00  Addition
  SUB  0x10  Subtraction
  MUL  0x20  Multiplication
  DIV  0x30  Division

  OR   0x40  Bitwise OR
  AND  0x50  Bitwise AND
  LSH  0x60  Left Shift
  RSH   0x70   Right Shift
  NEG   0x80   Negation

  (Jump, to an offset by the current instruction + JT/JF + 1)

  JA    0x00   Jump to the current instruction + K + 1
  JEQ   0x10   Jump if K or X equals accumulator
  JGT   0x20   Jump if K or X is greater than accumulator
  JGE   0x30   Jump if K or X is greater or equals the accumulator
  JSET  0x40   Jump if K or X bitwise AND the accumulator > 0

 Source:
  K     0x00   Value stored in K
  X     0x08   Value stored in the index register
```

RET specific fields:

```
 Return val:
  A     0x10   Value stored in the accumulator
  K     0x00   Value stored in K
  X     0x08   Value stored in the index register
```

The index register cannot use the packet addressing modes. Instead, a packet value must be loaded into the accumulator and transferred to the index register, via tax. This is not a common occurrence, as the index register is used primarily to parse the variable length IP header, which can be loaded directly via the 4*([k]&0xf) addressing mode.

MISC specific fields:

```
 Operation:
  TAX   0x00 Transfer value from accumulator into index register
  TXA   0x80 Transfer value from index register to accumulator
```

Example filter:

netsniff-ngs filter parser treats all lines that doesn't match a format of *{ 0xYY, X, X, 0xYYYYYYYY },* (X: decimal value, Y: hex value) as comments.

/etc/netsniff-ng/rules/arp.bpf:

```
1: { 0x28, 0, 0, 0x0000000c },
2: { 0x15, 0, 1, 0x00000806 },
3: { 0x06, 0, 0, 0xffffffff },
4: { 0x06, 0, 0, 0x00000000 },
```

The first instruction line is a load instruction, because we have LD|H|ABS which results in 0x28. So the 16 Bit valued halfword at the packet offset 0xc will be copied into the accumulator. This is the Ethernet type field. Instruction line 2 belongs to the class JMP, more specific JMP|JEQ and takes the value which is stored in K (0x806, the Ethernet type identifier for ARP). If 0x806 equals the value that has been loaded into the accumulator, the instruction pointer points to the current instruction plus jt value (which is 0) plus 1. So we end up at instruction line 3, which is the return opcode as RET|K. By using 0xffffffff as K, we tell the kernel that we would like to have a packet snaplen of 0xffffffff, which means that we end up with the complete (*uncut*) packet. 0xffffffff will be replaced by the real packet length if the kernel detects packets less than a length of 0xffffffff. Well, on the other hand we would trap into the jf value if we don't have an ARP packet. There, the instruction pointer will point to line 4 where we tell the kernel to drop the packet. The length of 0 simply means: Do not hand over the packet to the BPF attached socket.

In pretty-print this filter looks like:

```
(000) ldh      [12]
(001) jeq      #0x806          jt 2    jf 3
(002) ret      #-1
(003) ret      #0
```

Source: [1] http://www.tcpdump.org/papers/bpf-usenix93.pdf

BARE-METAL PERFORMANCE

This section will provide some figures about the performance of the RX_RING and TX_RING. An IBM HS21 Blade with 2 x Intel Xeon E5430 (2.66GHz), 8 GB RAM, Broadcom NetXtreme BCM5704S Gigabit Ethernet cards and a 2.6.31-14 kernel (Ubuntu Server 9.10) has been used for testing purpose. The IXIA 400 has been used on the opposite side for traffic generation (Gigabit wire speed). Date: 17 Sep 2010.

TX_RING, 1GbE:

The test was about flushing as much frames as possible of a fixed size. The IXIA was the counterpart that showed the incoming figures. Figures have been rounded to thousands.

```
Pkt size, TX_RING pps
   64     422,000
  128     422,000
  250     402,000
  500     239,000
  750     162,000
1,000     122,000
1,500      82,000
```

RX_RING, 1GbE:

The test included the reception of frames into the ring buffer, a counter increment per frame and the summation of the frame length. Figures have been rounded to thousands.

```
64-Byte fixed
Pkt rate (IXIA), % of BW, RX_RING pps
  100,000          6.75   100,000
  175,000         11.76   175,000
  250,000         16.80   250,000
  500,000         33.60   338,000
1,000,000         67.20   354,000
1,488,000        100.00   303,000

250-Byte fixed
Pkt rate (IXIA), % of BW, RX_RING pps
100,000           21.60   100,000
175,000           37.80   175,000
250,000           54.00   244,000
463,000          100.00   381,000

500-Byte fixed
Pkt rate (IXIA), % of BW, RX_RING pps
100,000           41.60   100,000
175,000           72.80   169,000
240,000          100.00   226,000

1,500-Byte fixed
Pkt rate (IXIA), % of BW, RX_RING pps
82,000           100.00   82,000

IMIX distribution (64:7, 570:4, 1518:1)
Pkt rate (IXIA), % of BW, RX_RING pps
100,000           29.99   100,000
175,000           52.35   175,000
250,000           74.80   240,000
334,000          100.00   303,000
```

```
Tolly distribution (64:55, 78:5, 576:17, 1518:23)
Pkt rate (IXIA), % of BW, RX_RING pps
100,000           40.50   100,000
175,000           70.90   174,000
247,000          100.00   193,000
```

EXAMPLES

Dump packets on eth0 into a file:

```
netsniff-ng --dev eth0 --dump out.pcap --silent --bind-cpu 0
```

Replay a PCAP file via eth0:

```
netsniff-ng --dev eth0 --replay out.pcap --bind-cpu 0
```

Only show ICQ related packets:

```
netsniff-ng --filter /etc/netsniff-ng/rules/icq.bpf
```

Show only packet headers of a PCAP file:

```
netsniff-ng --read out.pcap --no-payload
```

Show only packets that match a regular expression:

```
netsniff-ng --regex "foo.*bar"
```

Show only outgoing packets in hex format from wlan0:

```
netsniff-ng --dev wlan0 --all-hex --type outgoing
```

NOTES

If you try to create custom socket filters with *tcpdump -dd*, you have to edit the *ret* opcode of the resulting filter, otherwise your payload will be cut off:

0x6, 0, 0, 0xFFFFFFFF instead of 0x6, 0, 0, 0x00000060

The Linux kernel now takes skb→len instead of 0xFFFFFFFF. If you do not change it, the kernel will take 0x00000060 as buffer length and packets larger than 96 Byte will be cut off (filled with zero Bytes)!

Read http://dev.netsniff-ng.org/#4 for further technical details.

LICENSE

This program is distributed under the terms of the GNU General Public License as published by the Free Software Foundation. See COPYING for details on the License and the lack of warranty.

AVAILABILITY

The latest version of this program can be found at http://pub.netsniff-ng.org/netsniff-ng/.

BUGS

Bugs, what bugs? ;-) Okay, seriously . . .

The TX_RING is part of the kernel since 2.6.31. Needs kind of a compatibility mode for older kernels.

Currently, we don’t have a BPF compiler built-in, so that either the user needs to use filter definitions from /etc/netsniff-ng/rules, tcpdump -dd or write his own filter by hand.

Please send problems, bugs, questions, desirable enhancements, etc. to bugs@netsniff-ng.org.

GIT

git clone git://github.com/danborkmann/netsniff-ng.git

AUTHOR

netsniff-ng was originally written by Daniel Borkmann (daniel@netsniff-ng.org).

Current authors:

Daniel Borkmann (daniel@netsniff-ng.org), Emmanuel Roullit (emmanuel@netsniff-ng.org)

http://www.netsniff-ng.org/

The manpage has been written by Daniel Borkmann.

SEE ALSO

bpf(4), pcap(3), tcpdump(8)

IN HONOREM

To my alma mater:

> Leipzig University of Applied Science,
> Faculty of Computer Science, Mathematics and Natural Sciences