

Linux Kernel Backdoors And Their Detection

Joanna Rutkowska

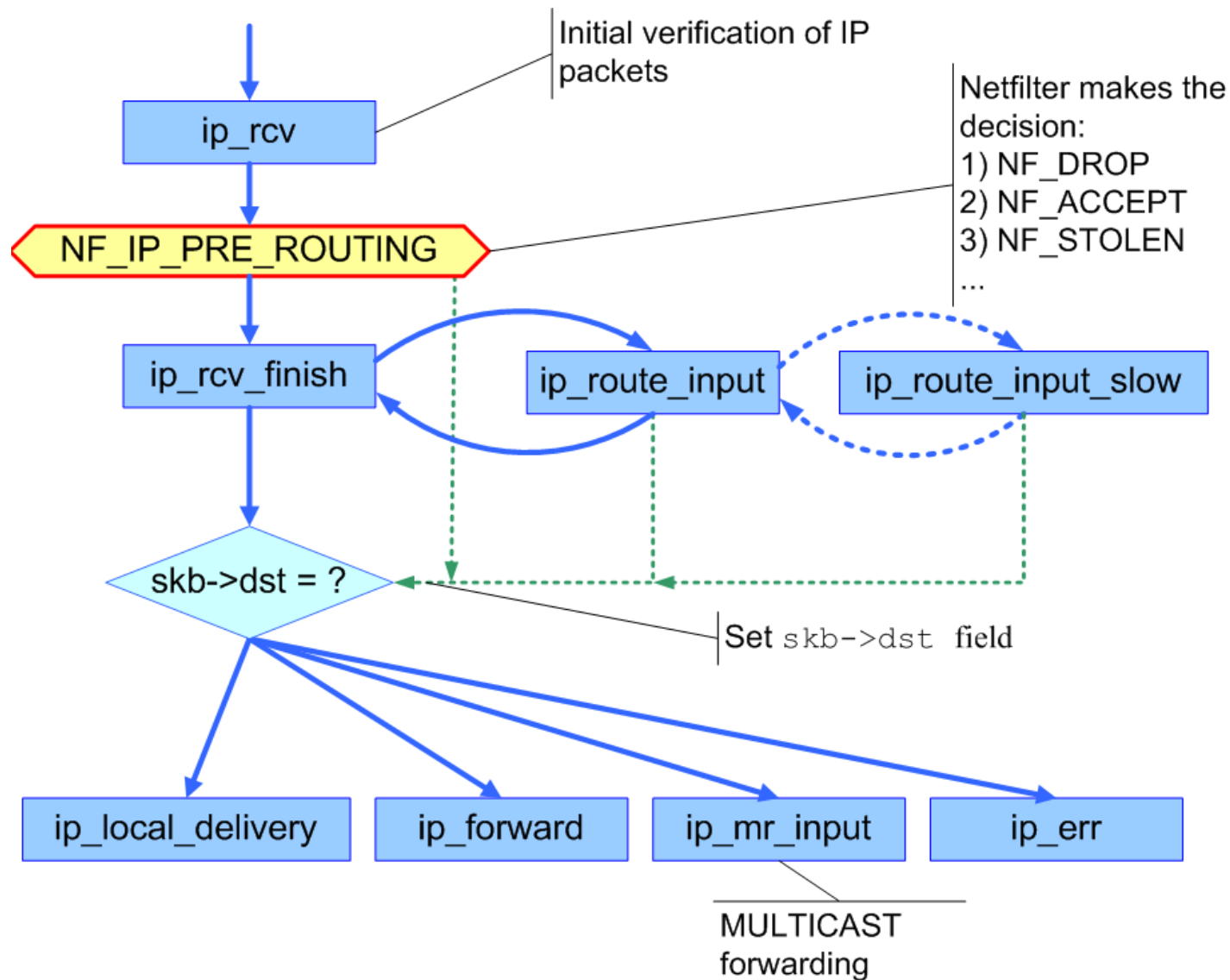
joanna@invisiblethings.org

ITUnderground Conference,
October 12th -13th 2004, Warsaw

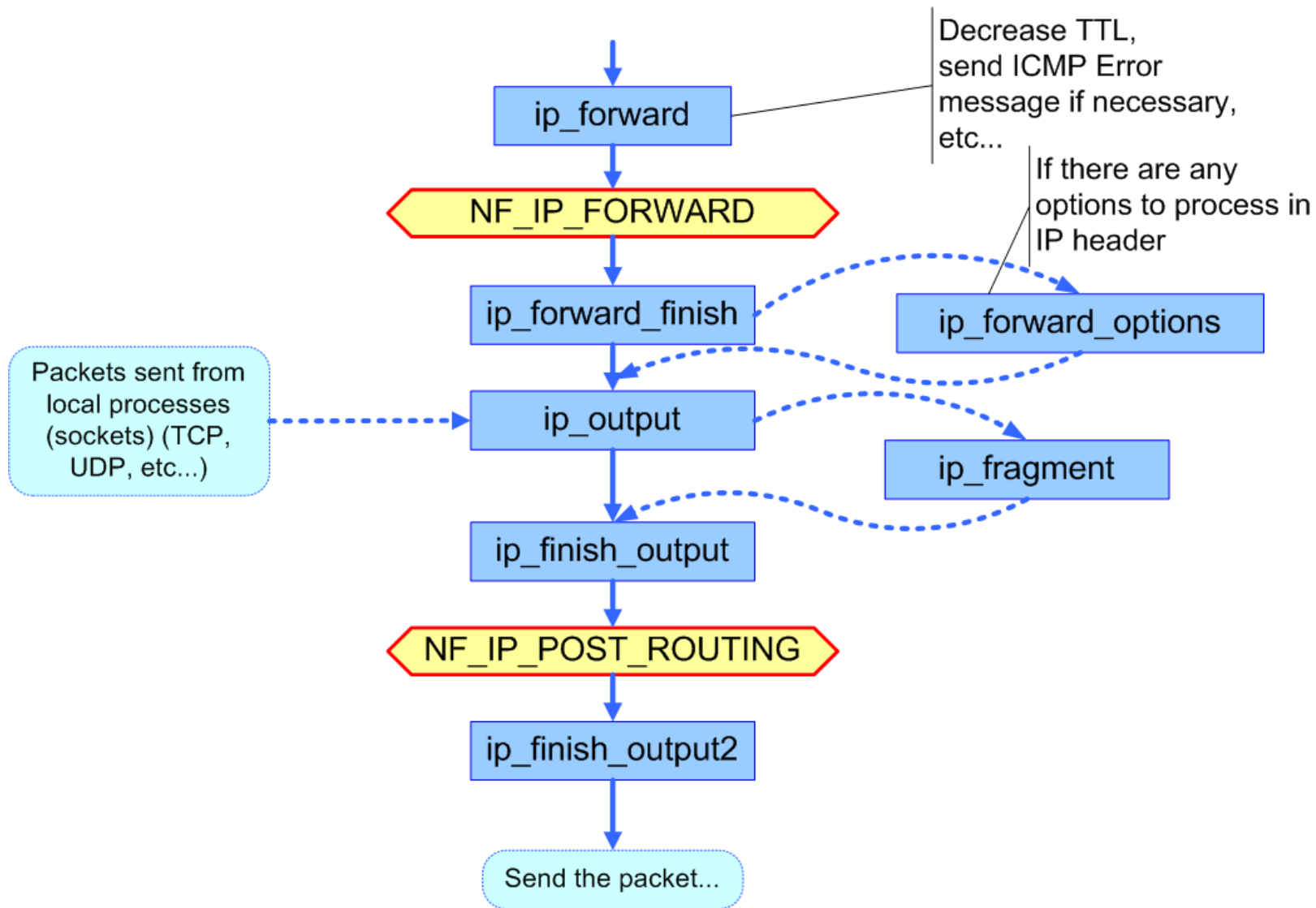
Goals

- ⊕ Provide some details about network kernel subsystem in modern Linux kernels,
- ⊕ Show potential techniques for implementing hard to detect network backdoors and covert channels,
- ⊕ Discuss possible approaches for detection of such malware.

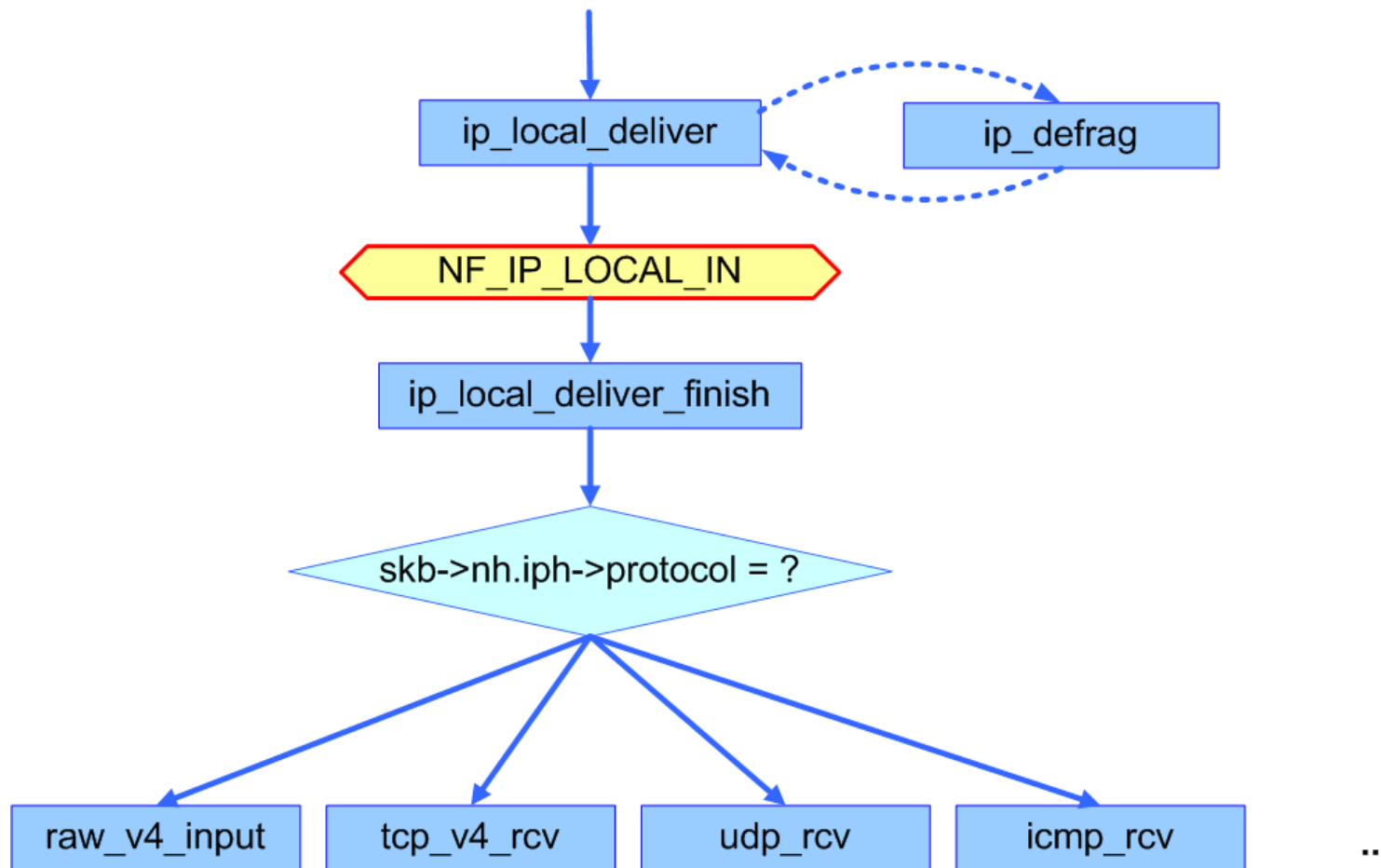
Network Packets Handling (2)



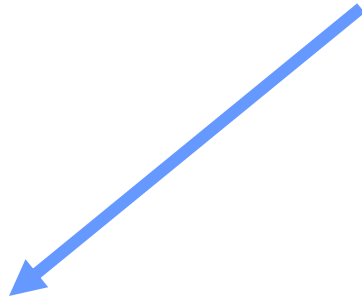
Network Packets Handling (3)



Network Packets Handling (4)

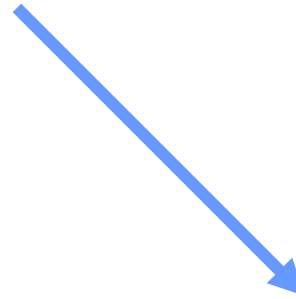


Two important techniques



`ptype_*` handlers

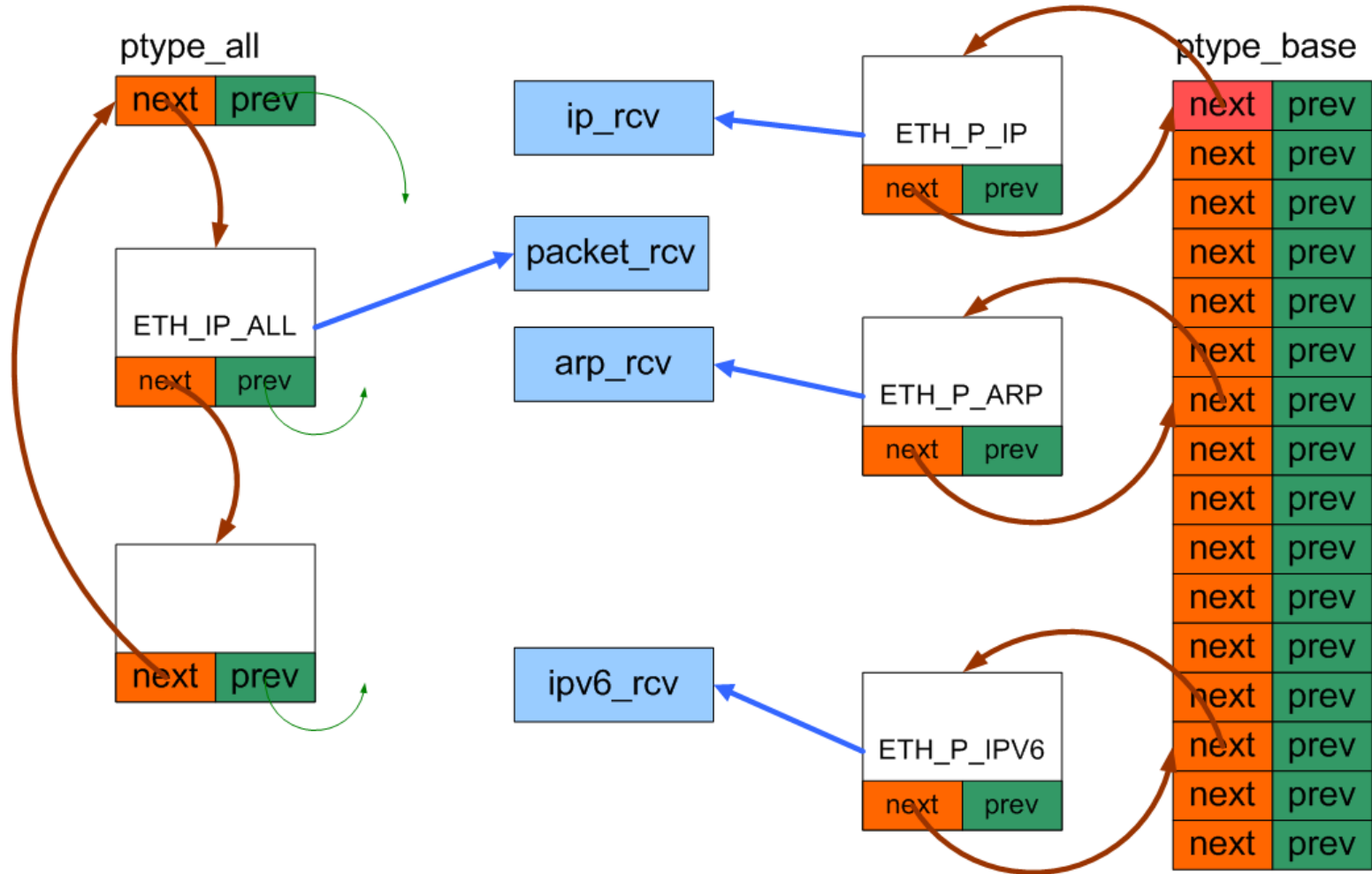
Add new protocol handler, which could simply recognize magic packets (SADoor-like) or modify packets' contents on the fly (passive covert channels)



Netfilter hooks

Add new NF hook, which could basically do the same things as ptype handler. Also it could circumvent local firewall (since ipfilter is also implemented using NF hooks).

protocol handlers



Key structure: *packet_type*

```
struct packet_type
{
    unsigned short    type; ← htons(ether_type)
    struct net_device *dev; ← NULL means all dev
    int (*func) (...); ← handler address
    void *data; ← private data
    struct list_head list;
};
```

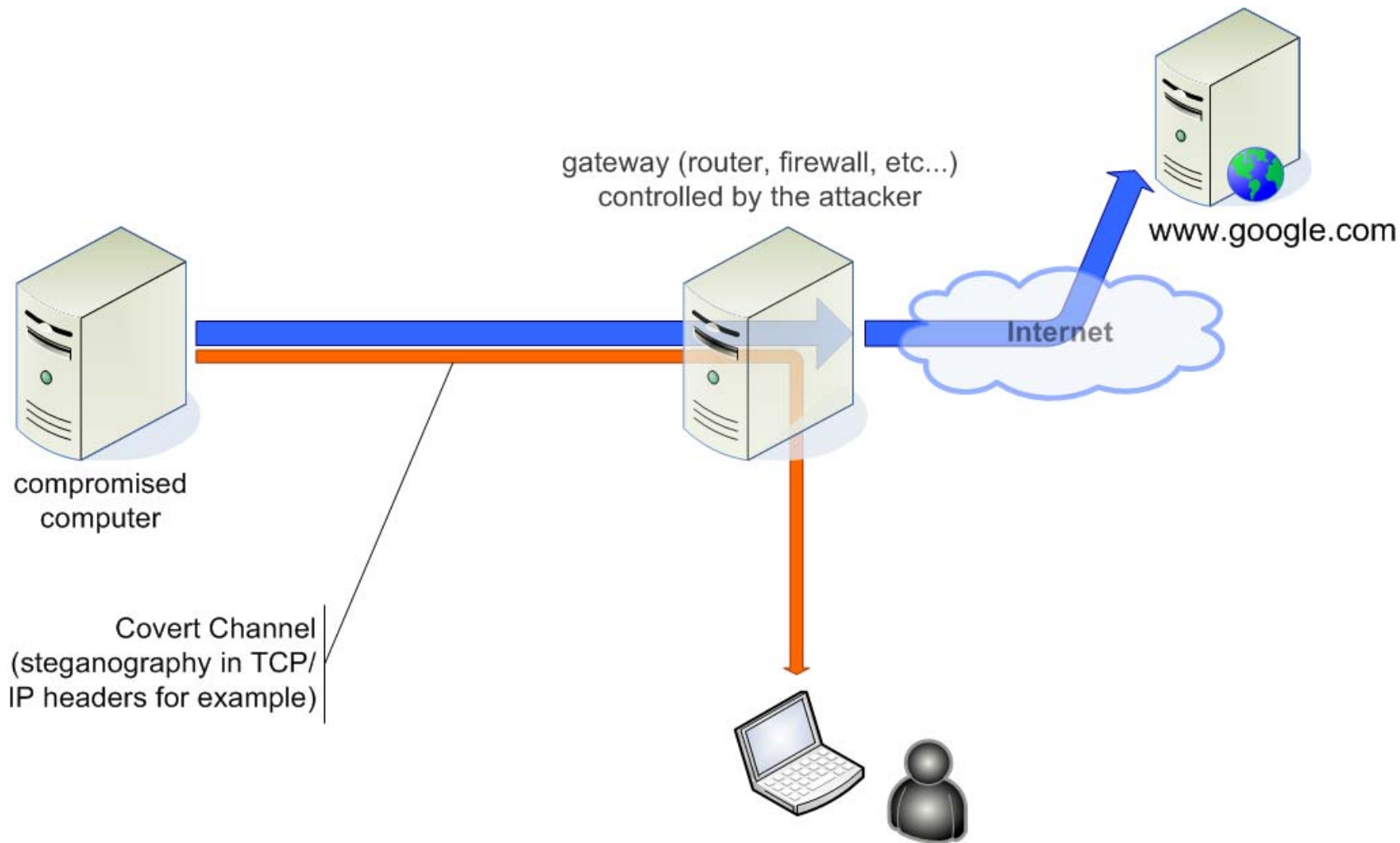
There are two exported kernel functions for adding and removing handlers:

- ⊕ **void** dev_add_pack(**struct** packet_type *pt)
- ⊕ **void** dev_remove_pack(**struct** packet_type *pt)

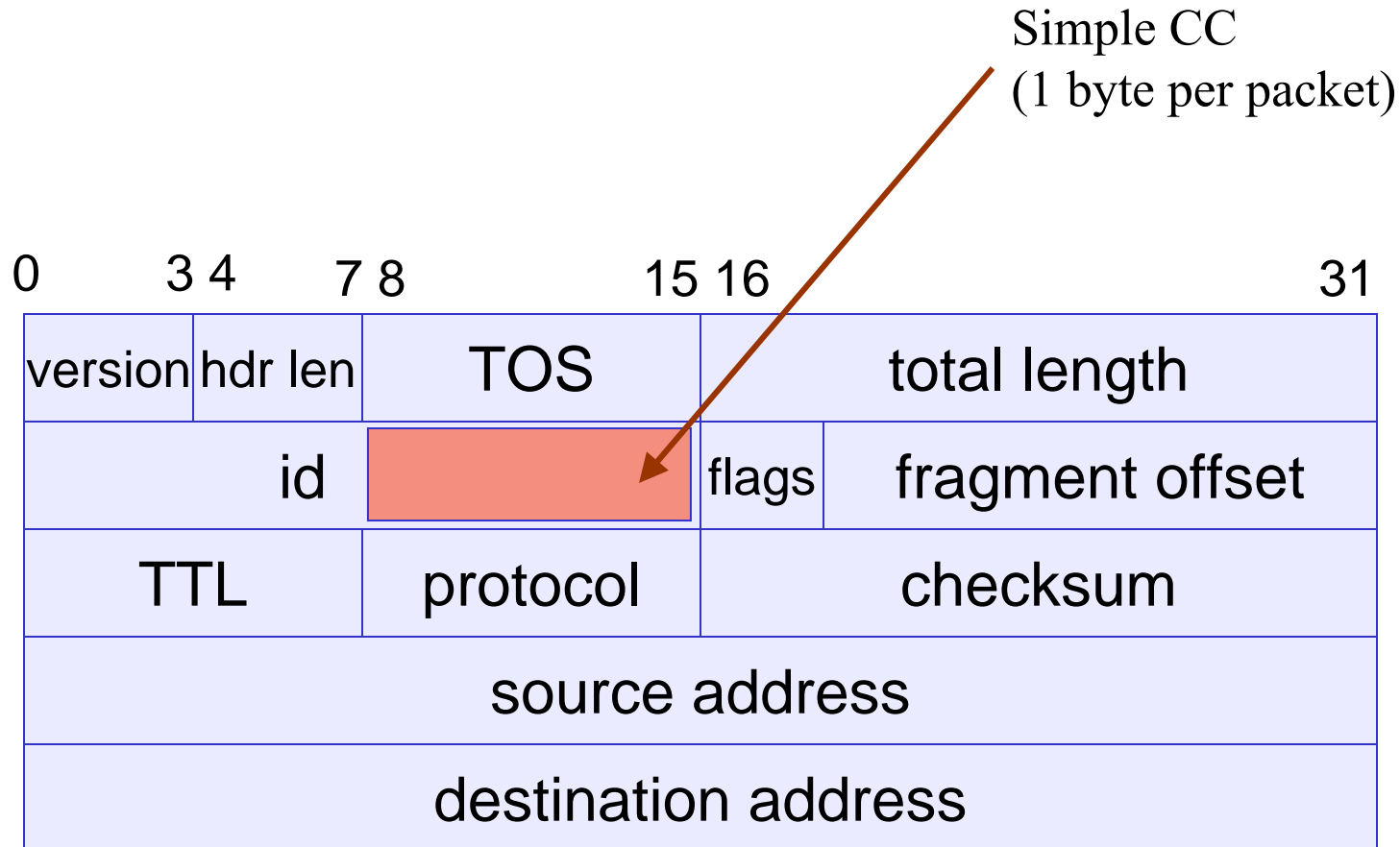
Addition of own handler

```
struct packet_type myproto;  
  
myproto.type = htons(ETH_P_ALL);  
myproto.func = myfunc;  
myproto.dev = NULL;  
myproto.data = NULL;  
  
dev_add_pack (&myproto)
```

Passive Covert Channel Example



IP Header and simple CC



DEMO: simple passive CC in IPID

secret message: **ABABA**

```
10.0.0.2 > 172.16.0.2: [[tcp] (DF) [tos 0x10] (ttl 64, id 16640, len 100)
    4510 0064 4100 4000 4006 4370 0a00 0002
    ac10 0002
```

```
10.0.0.2 > 172.16.0.2: [[tcp] (DF) [tos 0x10] (ttl 64, id 16896, len 52)
    4510 0034 4200 4000 4006 42a0 0a00 0002
    ac10 0002
```

```
10.0.0.2 > 172.16.0.2: [[tcp] (DF) [tos 0x10] (ttl 64, id 16640, len 52)
    4510 0034 4100 4000 4006 43a0 0a00 0002
    ac10 0002
```

```
10.0.0.2 > 172.16.0.2: [[tcp] (DF) [tos 0x10] (ttl 64, id 16896, len 100)
    4510 0064 4200 4000 4006 4270 0a00 0002
    ac10 0002
```

```
10.0.0.2 > 172.16.0.2: [[tcp] (DF) [tos 0x10] (ttl 64, id 16640, len 52)
    4510 0034 4100 4000 4006 43a0 0a00 0002
    ac10 0002
```

What is interesting in this simple CC?

- ⊕ Kernel implementation (as ptype handler)
- ⊕ There is no (easy) way to check what protocol handlers are registered in the kernel(!)
- ⊕ Seems like it is hard to detect that we have something like this in our kernel.
- ⊕ Of course you can detect *this* simple CC by network traffic analysis
- ⊕ We can imagine more complex CC implemented this way...

Simple places to hide information in HTTP protocol

```
GET http://www.somehost.com/cgi-  
bin/board.cgi?view=12121111 / HTTP/1.0  
Host: www.somehost.com  
User-Agent: Mozilla/5.0 (12121111)  
Accept: text/html  
Accept-Language: en,fr,en,fr,en,en,en,en  
Accept-Encoding: gzip,deflate,compress  
Accept-Charset: ISO-8859-1,utf-8,ISO-1212-1  
CONNECTION: close  
Proxy-Connection: close  
X-Microsoft-Plugin: unexpected error #12121111
```

source: <http://gray-world.net>

Digression about more interesting covert channels

- ⊕ TCP Timestamps: information hidden in least significant bit of TCP timestamp(!)
- ⊕ TCP ISN modulation. Note: unlike most current implementations, when considering passive ISN CC, the handler routine must track the whole TCP connection and change all the sequential numbers in all packets of that connection. It is necessary so that the user will not notice anything suspicious in her connection.
- ⊕ ???

All of the above ideas could be implemented as a passive channels using ptype handlers.

TCP Timestamp Covert Channel

- ✦ Idea and the simple implementation:
 - ✦ John Giffin i inni, *Covert Messaging Through TCP Timestamps*, Massachusetts Institute of Technology, 2002.
 - ✦ <http://www.mit.edu/~gif/covert-channel/>.
- ✦ Information hidden in LSB of TCP timestamp. Not as easy as it may sound – watch out for turning back in time!
- ✦ Secret shared by the two peers – without the secret it should be impossible to even find out that the communication occur (although the algorithm is known) – seems like real CC, based on theory and not on a “secret concept”.

How to detect?

- ⊕ How to get a list of registered protocol handlers?
- ⊕ Author does not know any tool (or even kernel API) for doing that!
- ⊕ We need to “manually” check the following lists:
 - ⊕ `ptype_all`
 - ⊕ `ptype_base`
- ⊕ But their addresses are not exported!

Where are the protocol lists?

- ⊕ Two kernel global variables (`net/core/dev.c`):
 - ⊕ `static struct packet_type *ptype_base[16];`
 - ⊕ `static struct packet_type *ptype_all = NULL;`
- ⊕ Only the following functions are referencing those variables (i.e. “know” their addresses):

<i>Kernel 2.4.20</i>	<i>Kernel 2.6.7</i>
<ol style="list-style-type: none">1. dev_add_pack()2. dev_remove_pack()3. <code>dev_queue_xmit_nit()</code>4. netif_receive_skb()	<ol style="list-style-type: none">1. dev_add_pack()2. __dev_remove_pack()3. dev_queue_xmit_nit()4. netif_receive_skb()5. <code>net_dev_init()</code>

The functions in **green** are exported.

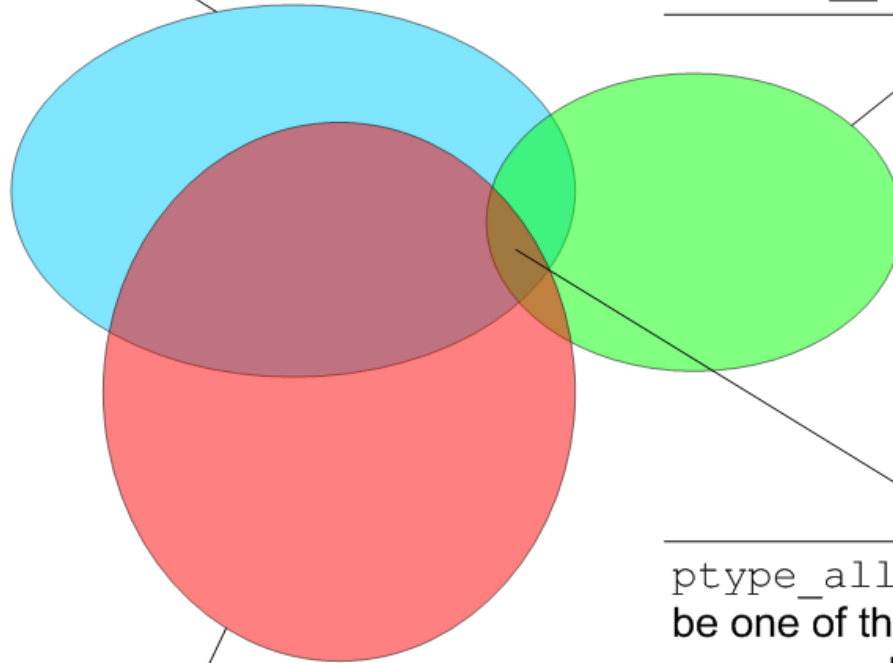
Approaches for finding the lists

- ⊕ `System.map` file
 - ⊕ problem: the file is not always up to date or sometimes it does not even exist (for security reasons;))
- ⊕ “heuristic” method
 - ⊕ We know the addresses of several functions which are using the addresses we are looking for.
 - ⊕ We can look at their body to find all the 32 bit words which *looks* like kernel pointers.
 - ⊕ Then we need to find the common set of those pointer-like words from all functions we consider.
 - ⊕ At last we need to check every potential value from the common subset to see if it *looks* like (or *could be*) `ptype_all` or `ptype_base` list head.

Illustration for heuristic method

Potential addresses obtained from
function `dev_add_pack()`

Potential addresses obtained from
function `__dev_remove_pack()`



`ptype_all` real address should
be one of the addresses from the
common subset (in practice it
contains about 3-4 addresses)

Potential addresses obtained from
function `netif_receive_skb()`

Live DEMO: detecting additional protocol handlers

PTYPE_ALL:

hook type ETH_P_ALL (0x3)

hook at: 0xc487e060 [module: **unknown module**]

PTYPE_BASE[]:

hook type ETH_P_IP (0x800)

hook at: 0xc0203434 -> ip_rcv() [k_core]

hook type ETH_P_802_2 (0x4)

hook at: 0xc01f8050 [k_core]

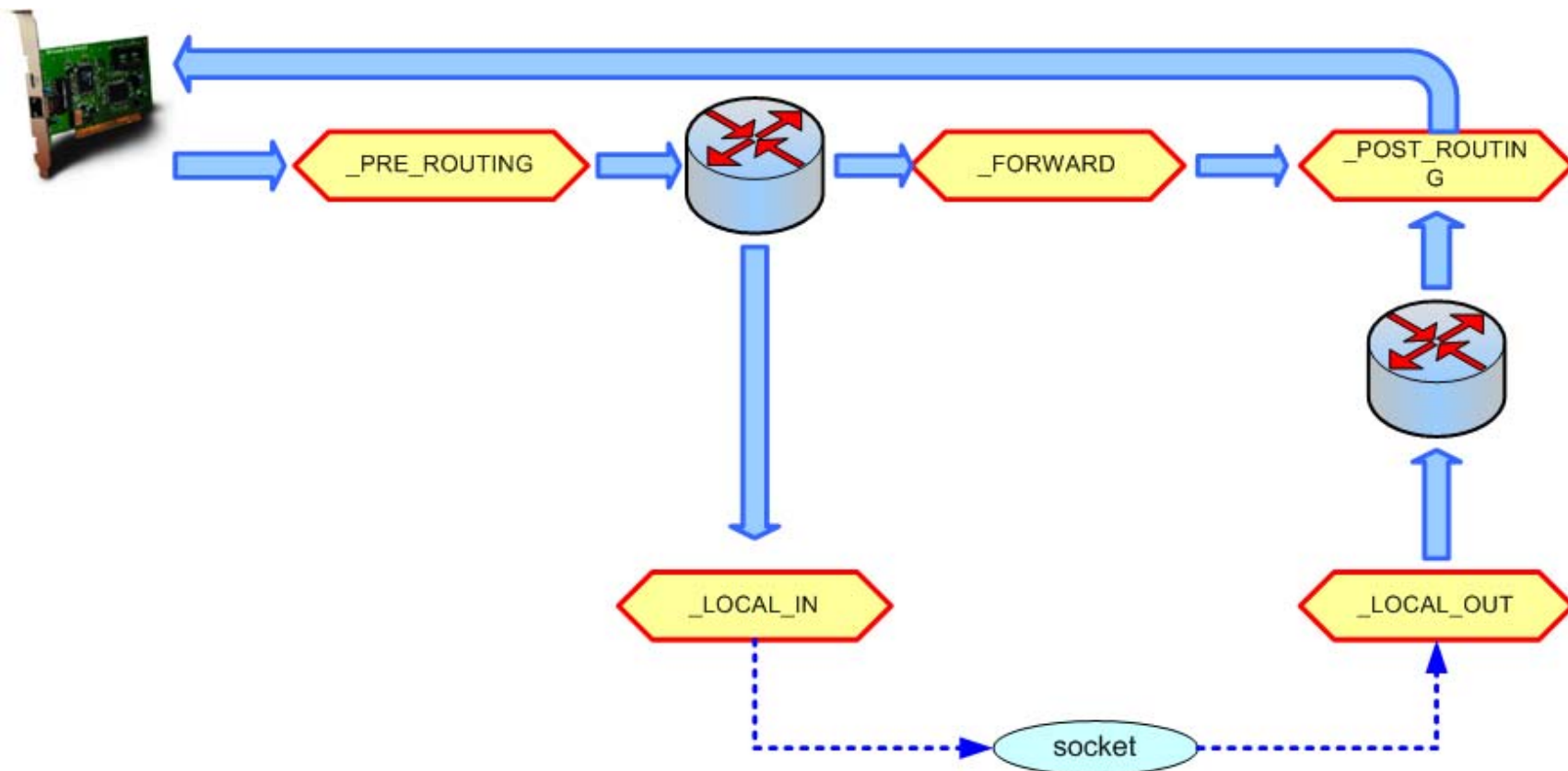
hook type ETH_P_ARP (0x806)

hook at: 0xc0223778 -> arp_rcv() [k_core]

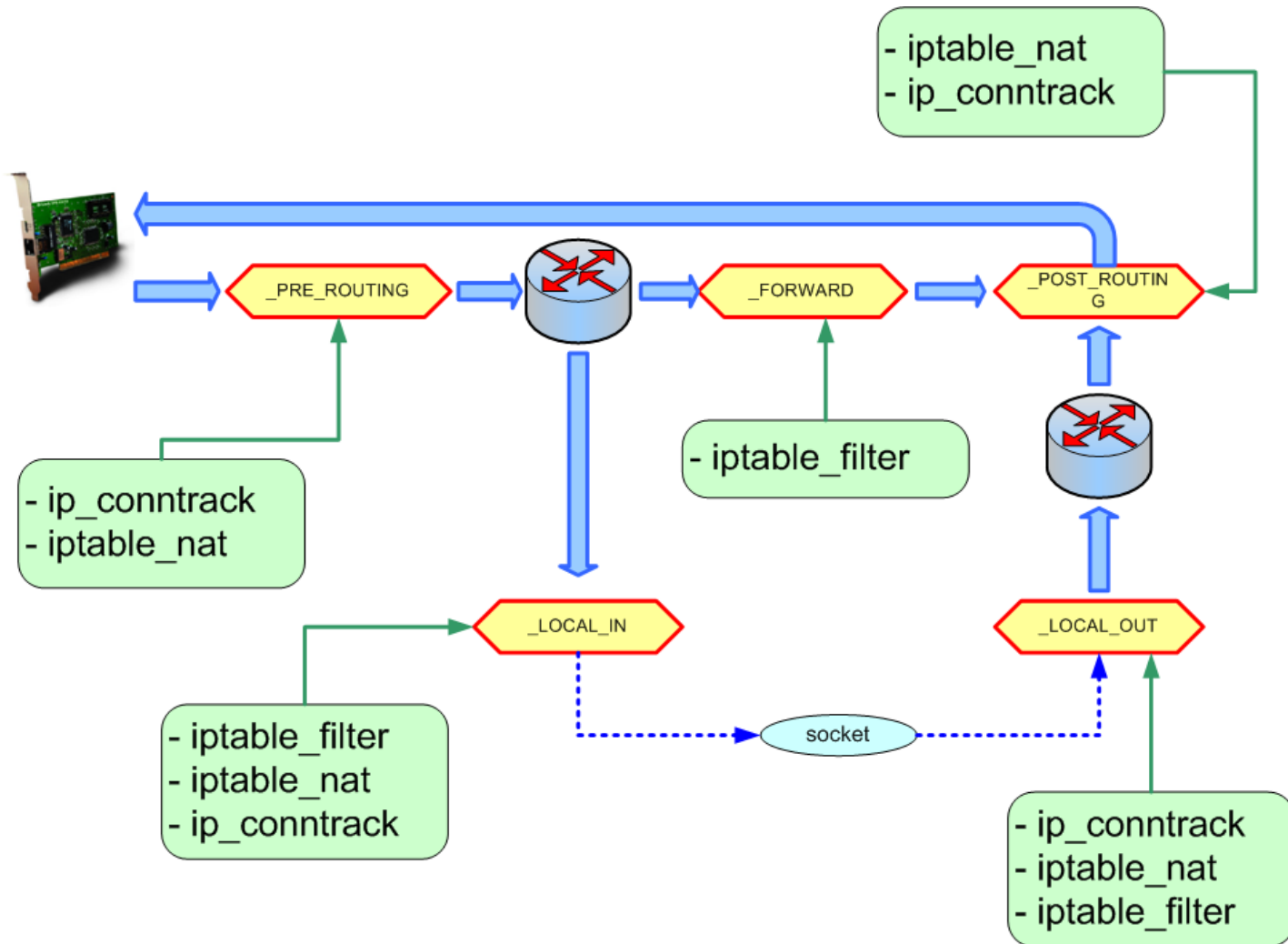
Lets now talk about Netfilter hooks...

- ⊕ Another way of implementing smart backdoors or CCs in kernel is to register NF hook.
- ⊕ We've seen already some NF hooks on the network map.
- ⊕ What is Netfilter exactly?

NF HOOKS



NF HOOKS: iptables implementation



Key structure: *nf_hook_ops*

```
struct nf_hook_ops
{
    struct list_head list;
    nf_hookfn *hook; ← handler address
    int pf; ← family (AF_INET, AF_INET6, AF_IPX, ...)
    int hooknum; ← (NF_IP_*, NF_IP6_*, NF_IPX_*, ...)
    int priority;
};
```

There are two exported kernel functions for adding and removing handlers:

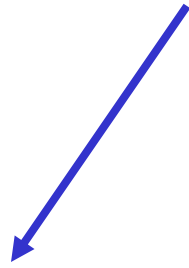
- ⊕ **int** nf_register_hook(**struct** nf_hook_ops *reg);
- ⊕ **void** nf_unregister_hook(**struct** nf_hook_ops *reg);

Adding new NF hook

```
myhook.hook      = myhandler;  
myhook.pf        = PF_INET;  
myhook.priority  = NF_IP_PRI_FIRST;  
myhook.hooknum   = NF_IP_POST_ROUTING;  
  
nf_register_hook(&myhook);
```

Registered NF hooks lists

```
struct list_head  
    nf_hooks[NPROTO][NF_MAX_HOOKS];
```



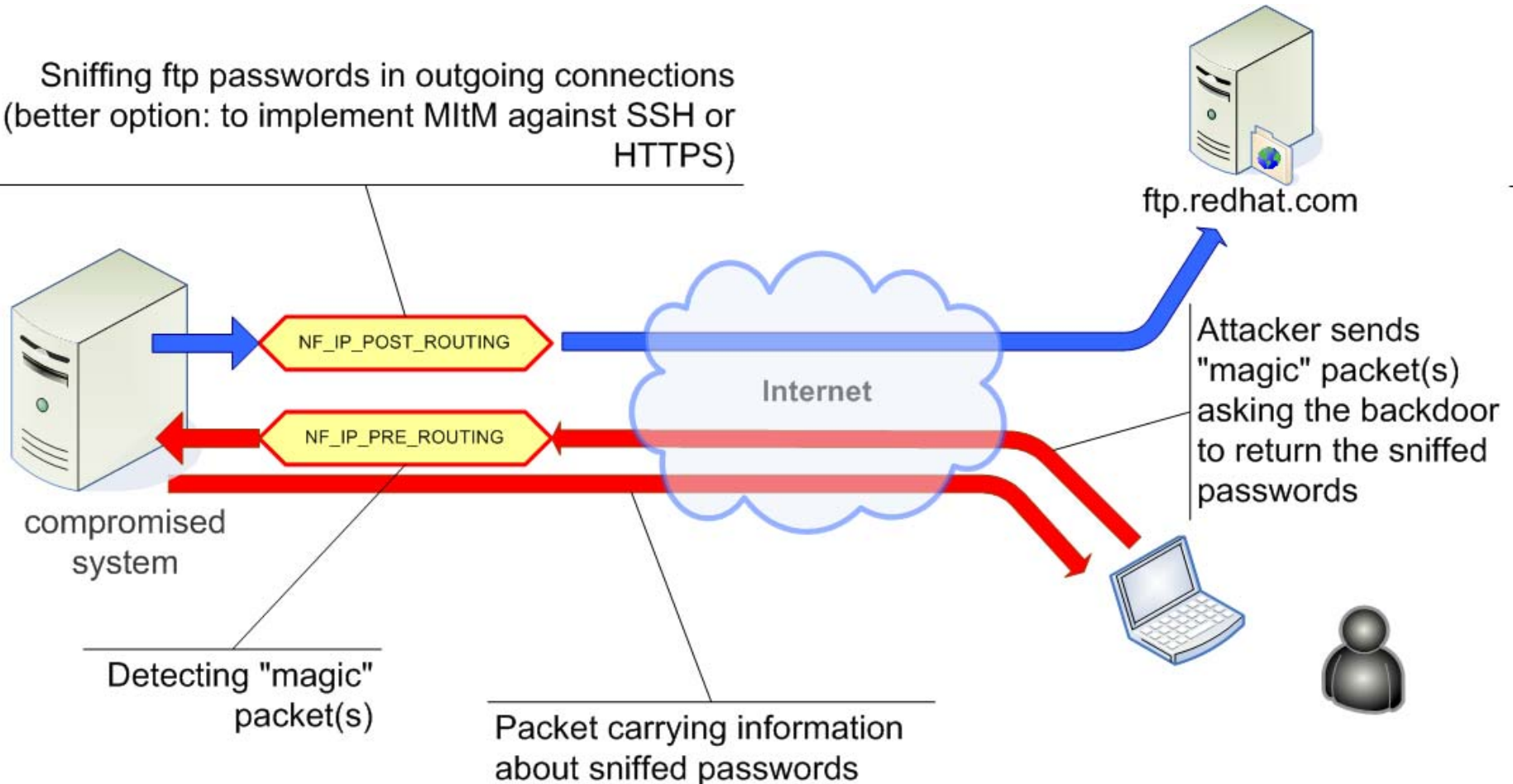
AF_UNSPEC (0)
AF_UNIX (1)
AF_INET (2)
AF_AX25 (3)
AF_IPX (4)
AF_APPLETALK(5)



{
NF_IP_PRE_ROUTING (0)
NF_IP_LOCAL_IN (1)
NF_IP_FORWARD (2)
NF_IP_LOCAL_OUT (3)
NF_IP_POST_ROUTING(4)

Simple example

Sniffing ftp passwords in outgoing connections
(better option: to implement MITM against SSH or HTTPS)



implementation: bioforge, *Hacking the Linux Kernel Network Stack*, Phrack Magazine, Issue 61.

How to detect?

- ✦ Fortunately `nf_hooks` address is exported by all kernels (known to the author) :)
- ✦ Knowing the address of `nf_hooks` we just need to scan all the hook lists and find the suspicious hooks.
- ✦ Suspicious are for example the ones, which do not belong to any modules address space (since the modules have been hidden or removed by the attacker or the code has been injected through `/dev/(k)mem`)

Live DEMO: Enumerating Netfilter hooks

-- NETFILTER hooks ---

INET NF_IP_PRE_ROUTING hooks:

hook at: 0xc48583ca [module: unknown module]

INET NF_IP_LOCAL_IN hooks:

hook at: 0xc4854060 [module: iptable_filter]

INET NF_IP_FORWARD hooks:

hook at: 0xc4854060 [module: iptable_filter]

INET NF_IP_LOCAL_OUT hooks:

hook at: 0xc485409c [module: iptable_filter]

INET NF_IP_POST_ROUTING hooks:

hook at: 0xc485834a [module: unknown module]

NF Hooks vs. ptype handlers

- ⊕ Both techniques allows attacker to implement kernel backdoors quite easily
- ⊕ It is much harder to detect backdoors implemented as ptype handlers then NF hooks – we need heuristics to find ptype lists (as it was showed)
- ⊕ NF Hooks on the other hand have got some potential advantages of bypassing local IP filters.

Other techniques for implementing backdoors

- ⊕ Raw sockets
 - ⊕ AF_PACKET (used also by `tcpdump`)
 - ⊕ AF_INET RAWIP (used also by `ping`)
- ⊕ (Raw) socket can be created by
 - ⊕ usermode process → we need to hide process & socket
 - ⊕ kernel code → we need to hide socket
- ⊕ Socket hiding against: `proc/net/raw`, `/proc/net/packet`
- ⊕ Passive CC cannot be implemented using (raw) sockets.

References

- ✦ Linux Kernel Sources, <http://kernel.org>
- ✦ Netfilter Official Documentation, <http://netfilter.org>
- ✦ kossak, *Building Into The Linux Network Layer*, Phrack Magazine, Issue 55, <http://phrack.org>
- ✦ bioforge, *Hacking the Linux Kernel Network Stack*, Phrack Magazine, Issue 61, <http://phrack.org>
- ✦ Craig H. Rowland, *Covert Channels in the TCP/IP Protocol Suite*, First Monday, 1996, http://www.firstmonday.dk/issues/issue2_5/rowland/.
- ✦ John Giffin i inni, *Covert Messaging Through TCP Timestamps*, Massachusetts Institute of Technology, 2002.
- ✦ John Giffin, prosty program implementujący ukryty kanał w znacznikach czasowych TCP, <http://www.mit.edu/~gif/covert-channel/>.
- ✦ Andrew Hintz, *Covert Channels in TCP and IP Headers*, 2003, <http://guh.nu/projects/cc/covertchan.ppt>.

The End