(kernel)(Hacking)

if i really know, i can hack

Kernel x Segurança

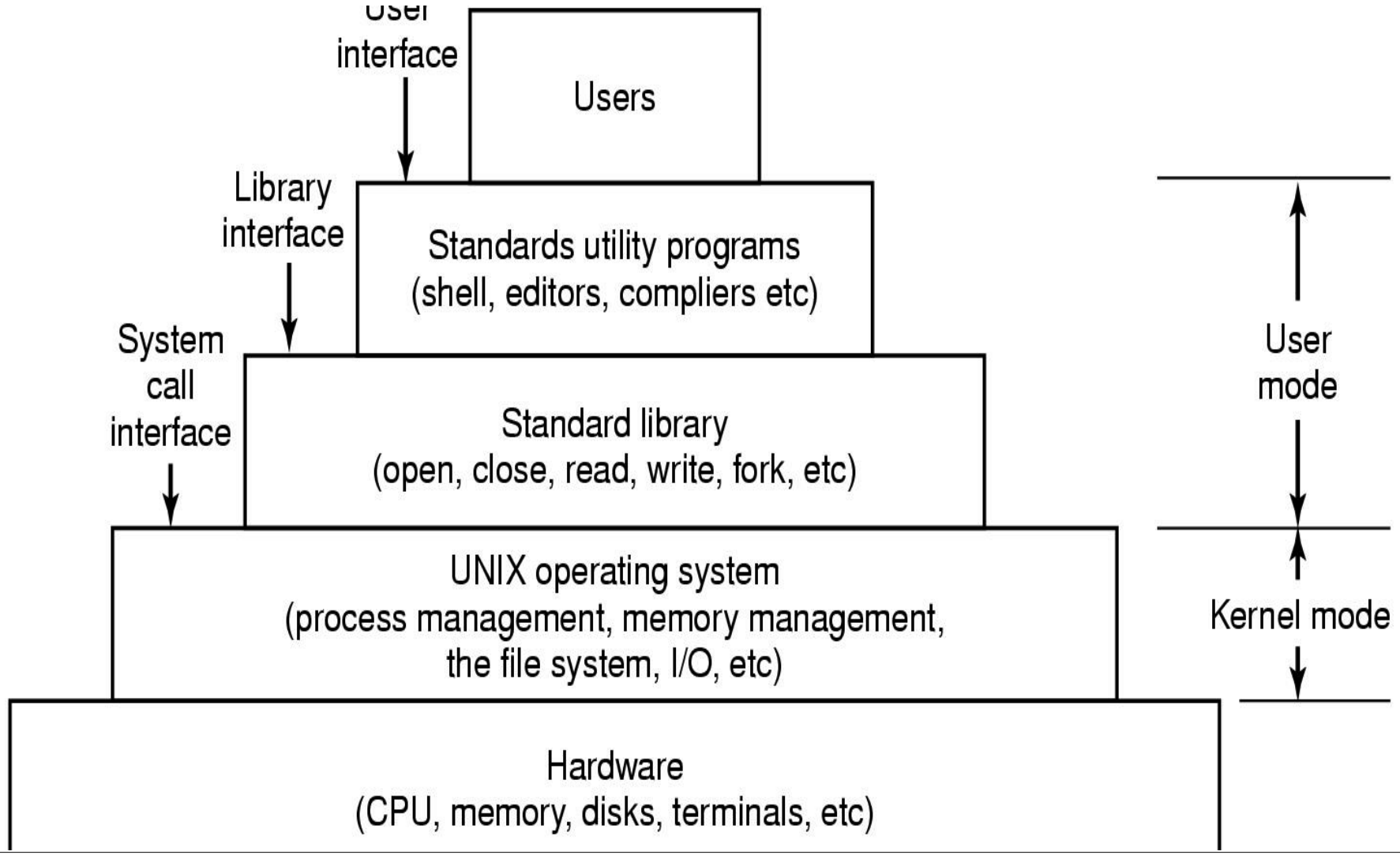
Rodrigo Rubira Branco

rodrigo@kernelhacking.com
bsdaemon@bsdaemon.org

OBJETIVOS

- Demonstrar recursos existentes no kernel do Linux
- Explicar estruturas internas do kernel do Linux
- Explicitar possíveis melhorias de seguranca para o kernel
- Exemplificar recursos de segurança e de ataques
- Exemplificar recursos de módulos / injeção de código em kernel

GNU/Linux


User interface
Users
Library interface
Standards utility programs
(shell, editors, compliers etc)
System call interface
Standard library
(open, close, read, write, fork, etc)
UNIX operating system
(process management, memory management,
the file system, I/O, etc)
Hardware
(CPU, memory, disks, terminals, etc)
User mode
Kernel mode

STDLIB (Standard Library)

- Chamadas de sistema são os métodos utilizados por programas de usuário para pedirem ações do sistema operacional
- Esta chamada é efetuada através de um trap:
  - Mudança do modo usuário para modo kernel
  - Controle dos parâmetros corretos
  - Execução efetuada pelo sistema operacional
  - Retorno ao modo usuário
- Como é impossível executar um trap em C, este é provido pela standard lib através de funções para cada system call. Estas funções são escritas em ASM e então chamadas a partir do C.
- A stdlib define um conjunto de funções através das quais as aplicações podem interagir com o kernel, e implementa muitas funcionalidades do sistema operacional que não necessitam da interação do kernel
- O padrão POSIX estabelece quais são as funções da biblioteca que o sistema necessita providenciar, seus parâmetros e tarefas.

Kernel

- Responsável por manter as importantes abstrações do sistema operacional
- O kernel provê as principais funções da “máquina abstrata” (system calls, interrupções e traps).

Estrutura Genérica de um Kernel Unix

<table>
<tr><td colspan="5">System calls</td><td colspan="3">Interrupts and traps</td></tr>
<tr><td colspan="2">Terminal handing</td><td>Sockets</td><td>File naming</td><td>Map-ping</td><td>Page faults</td><td rowspan="2">Signal handling</td><td rowspan="2">Process creation and termination</td></tr>
<tr><td rowspan="2">Raw tty</td><td>Cooked tty</td><td>Network protocols</td><td>File systems</td><td colspan="2">Virtual memory</td></tr>
<tr><td>Line disciplines</td><td>Routing</td><td>Buffer cache</td><td colspan="2">Page cache</td><td colspan="2">Process scheduling</td></tr>
<tr><td colspan="2">Character devices</td><td>Network device drivers</td><td colspan="2">Disk device drivers</td><td colspan="2">Process dispatching</td></tr>
<tr><td colspan="8">Hardware</td></tr>
</table>

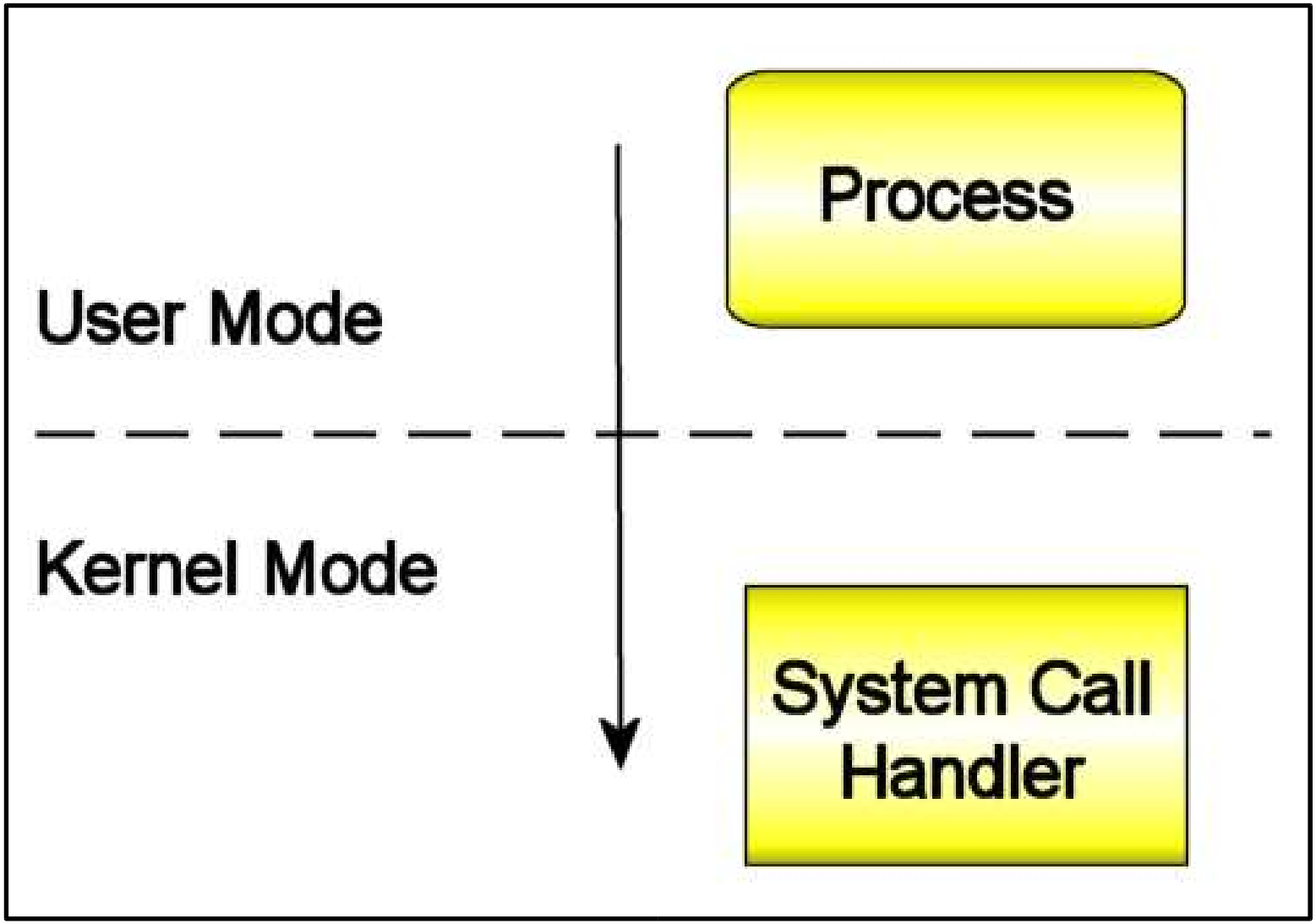
Kernel

- O Kernel do Linux possui um modelo monolítico e não o "novo" modelo cliente-servidor (ex: micro-kernels, como minix). Isto por motivos de performance.
  - O kernel é executado em modo kernel, com total controle aos recursos físicos do computador.
  - Todo o código do kernel e os dados são mantidos em um único espaço de endereços.
- Embora o kernel execute como um simples processo em um único espaço de endereçamento, este possui modularidade.

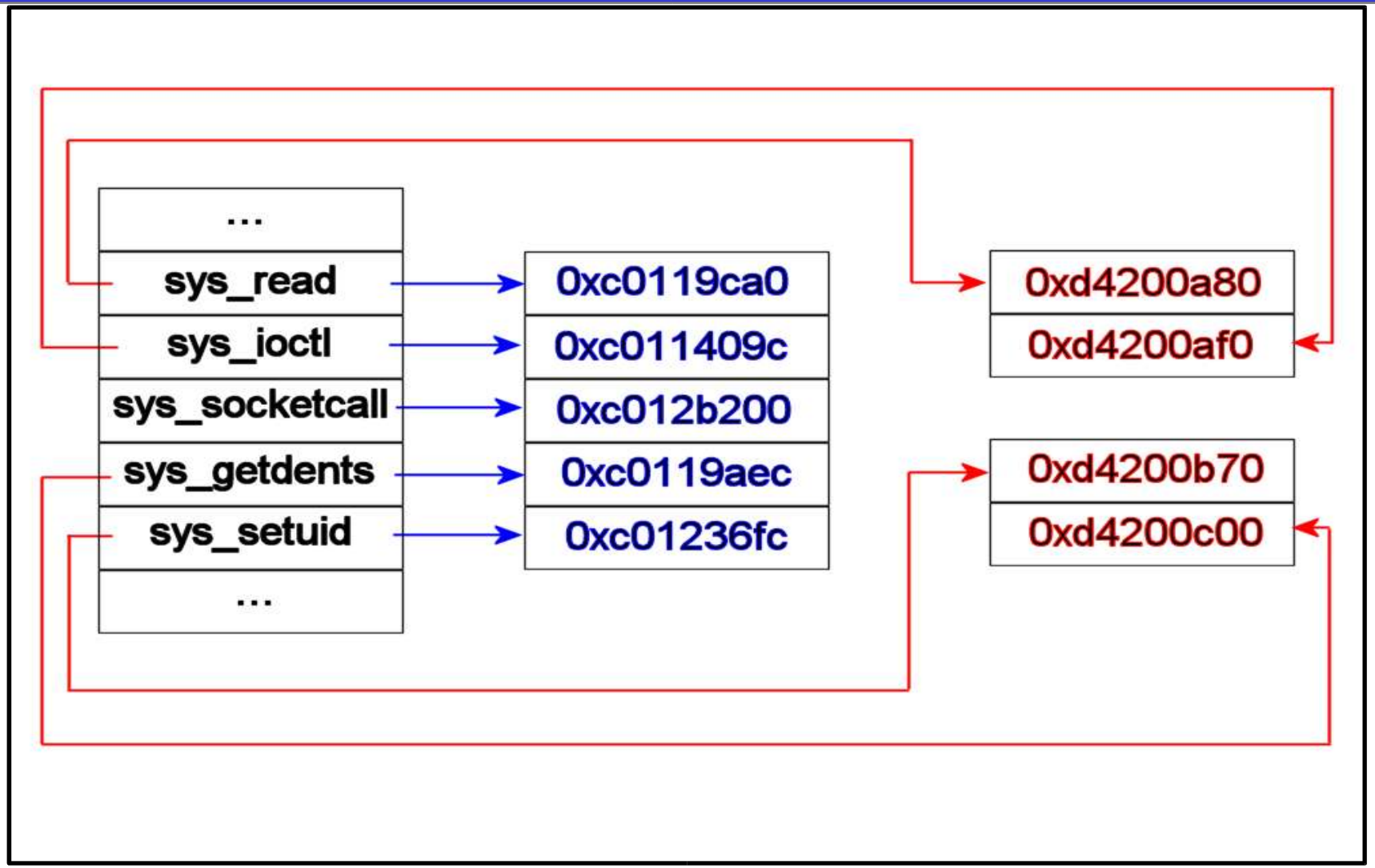
Problemas com a Segurança Atual

- Evitar / Identificar / Corrigir
- O estado atual da segurança é uma corrida sem fim
- Ciclo sem fim de descobrimento/correção de vulnerabilidades

Syscalls (chamadas ao sistema)



Syscalls (chamadas ao sistema)



Syscalls (chamadas ao sistema)

```
extern void *sys_call_table[];
int (*o_getdents) (uint, struct dirent *, uint);

o_getdents=sys_call_table[SYS_getdents];
sys_call_table[SYS_getdents]=(void*)h_getdents;
```

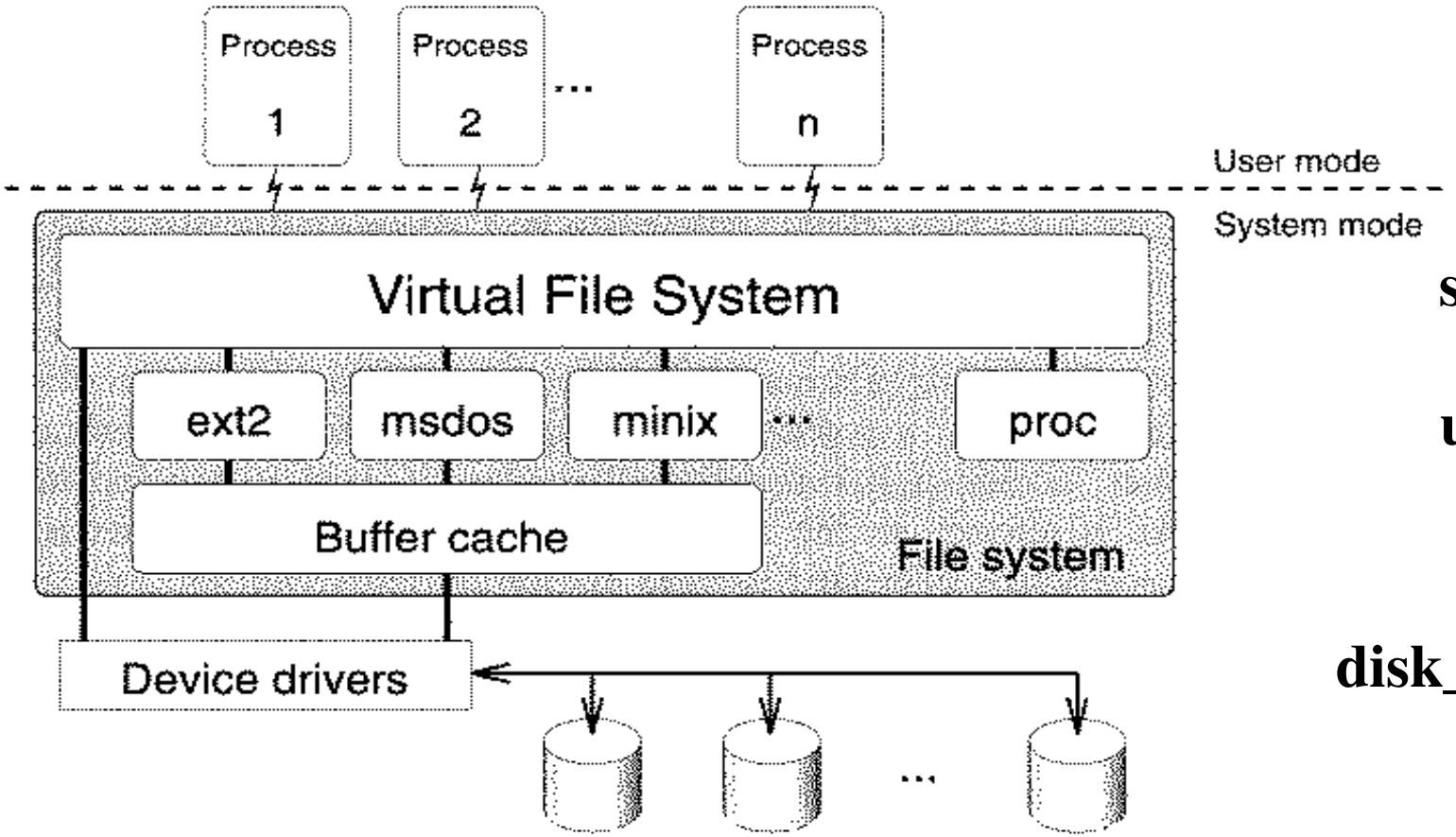
Syscalls (chamadas ao sistema)

```
void *hacked_sys_call_table;
hacked_sys_call_table=kmalloc(256*sizeof(long int),
    GFP_KERNEL);
memcpy(hacked_sys_call_table, sys_call_table, 256*sizeof(long
    int));

(int)*((int*)ptr) =(int) hacked_sys_call_table;
```

- PTR aponta para o endereço original da tabela (chamada pela função system_call)

VFS (Virtual File System)



read()

sys_read()

ufs_read()

disk_dev_read()

VFS – Exemplo de Hooks (chamadas /proc)

- Interceptando sem o uso das syscalls

```
old_readdir_root = proc_root.FILE_OPS->readdir;
old_lookup_root = proc_root.INODE_OPS->lookup;
proc_root.FILE_OPS->readdir = &new_readdir_root;
proc_root.INODE_OPS->lookup = &new_lookup_root;
```

VFS Security

- Mecanismos de segurança impostos pelo VFS e alguns file systems:
  - Mount flags (super_block→s_flags)
  - File attributes (inode→i_mode)
  - Extended attributes (Kernel 2.6)
  - File ownership (inode→u_uid)
  - Restrições adicionais aos modelos existentes:
    - Atributos extras ext2, ext3
    - Append-Only, Immutable

VFS - Fraquezas

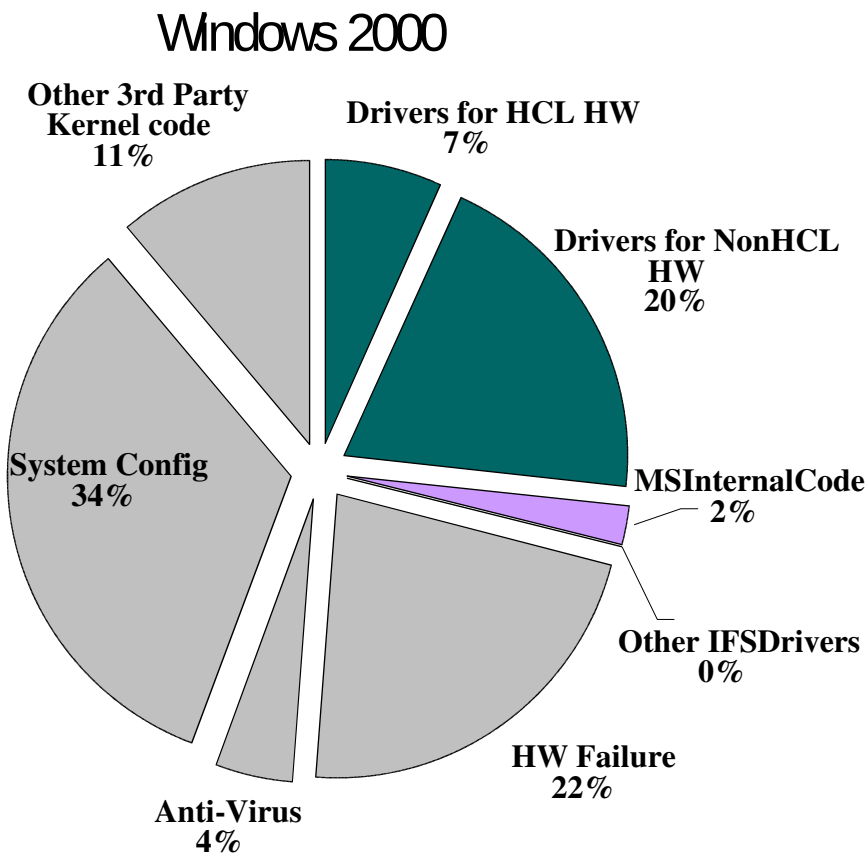
Um atacante que ganhe acesso root, possui completo controle sobre o sistema de arquivos

- Apagar entradas em logs
  - Ext2 append-only flag não impede o usuário root
- Insmod
  - Checagens de segurança do VFS/sistema de arquivos podem ser passadas
    - Interceptando-se chamadas de sistema ao VFS
    - Substituindo-se operações nas tabelas linkadas na criação de arquivos/inodes
- Comunicando-se diretamente com o device driver (/dev)

Drivers de Dispositivos e Problemas de Segurança

- Windows 2000: #1 em bugs de kernel reportados [Murphy '00]

➢ Os drivers não são controlados pelos Sistemas Operacionais, afetando e muito a segurança dos mesmos



Source: Brendan Murphy, Sample from PSS Incidents

Drivers de Dispositivos e Problemas de Segurança

- Podemos aumentar a segurança do SO através de “tolerância” de erros de drivers de dispositivos
- Necessita manter-se compatibilidade com os drivers existentes

➢ Possível Solução: Isolar os device drivers em uma “sandbox”, mantendo as APIs existentes

Drivers de Dispositivos e Problemas de Segurança

- Performance do isolamento depende:
  - Nível do isolamento requerido
  - Custo das comunicações entre isolamentos
  - Custo das movimentações de dados entre os isolamentos
  - Custo da execução do código isolado

➢ Problema: Necessita-se conhecimento dos drivers antes de se isolar os mesmos.

Drivers de Dispositivos e Problemas de Segurança

- Diferenças ao lidarmos com drivers:
  - Já existem!
  - Lidam com quantidade absurda de dados
  - Possuem apelas recursos limitados de aplicação

Drivers de Dispositivos e Problemas de Segurança

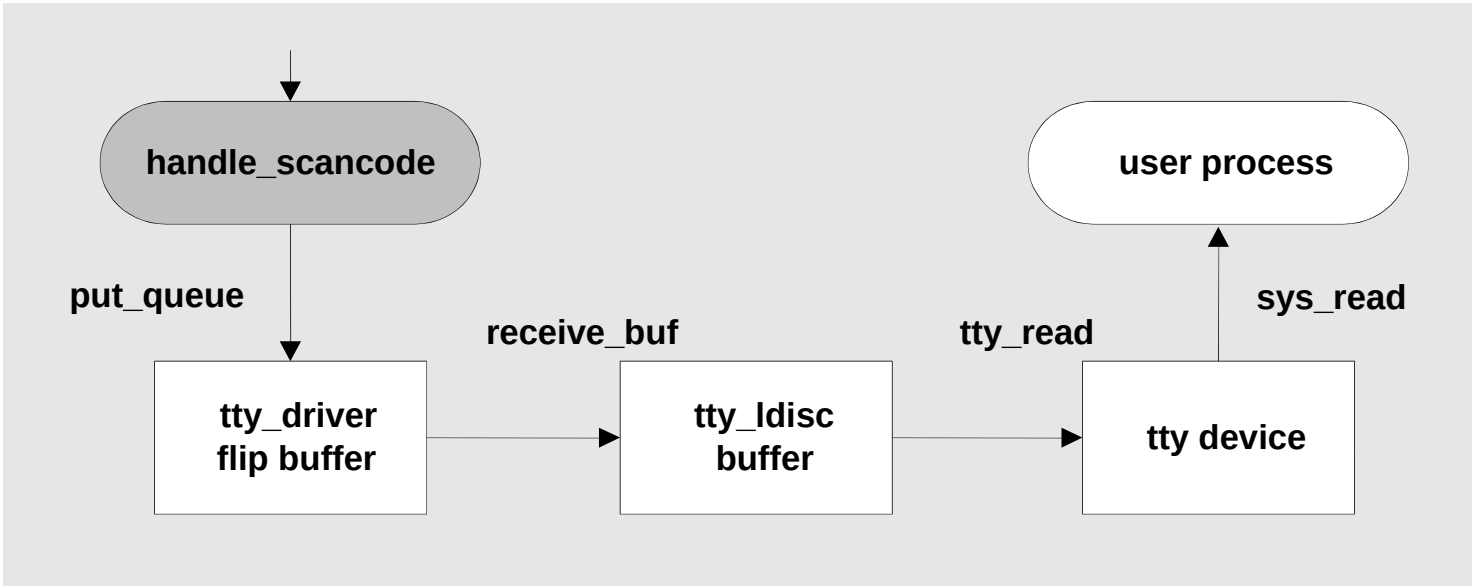
- A maioria dos problemas em drivers são derivados de: [Chou '01, Linux kernel Bugzilla]
  - Acesso ilegal a memória
  - Uso inválido de locks
  - Mantimento de interrupções desabilitadas

- Estes problemas podem ser detectados verificando-se os acessos a memória e as condições anteriores/posteriores na execução do driver.

Drivers de Dispositivos e Problemas de Segurança

- Kernel já suporta inicialização/paralização de drivers dinamicamente
- Devido ao grande número de interfaces driver/kernel no Linux, temos muitas oportunidades de otimização:
  - Muitos parâmetros read-only
  - Maioria das funções são de inicialização/limpeza

TTY Sniffer


handle_scancode
user process
put_queue
receive_buf
tty_read
sys_read
tty_driver
flip buffer
tty_ldisc
buffer
tty device

TTY Sniffer

- O modo comum de se desenvolver keyloggers para kernel é
- interceptar chamadas ao sistema read/write para armazenar as
- informações passadas a estas

  - Permite logar dados de todas as seções de usuários
  - Diminui performance de todo o sistema
  - Fácil de detectar (verificando-se as syscalls)

TTY Sniffer

- Interceptação: receive_buf
  - receive_buf é chamada pelo driver de baixo nível do tty

```
/* drivers/char/n_tty.c */
static void n_tty_receive_buf(struct tty_struct *tty, const
                  unsigned char *cp, char *fp, int count)
```

  - Para interceptar esta função, basta apontar tty_ldisc.receive_buf()
  - A função hackiada tem como objetivo logar as entradas de usuário.

TTY Sniffer

- Exemplo

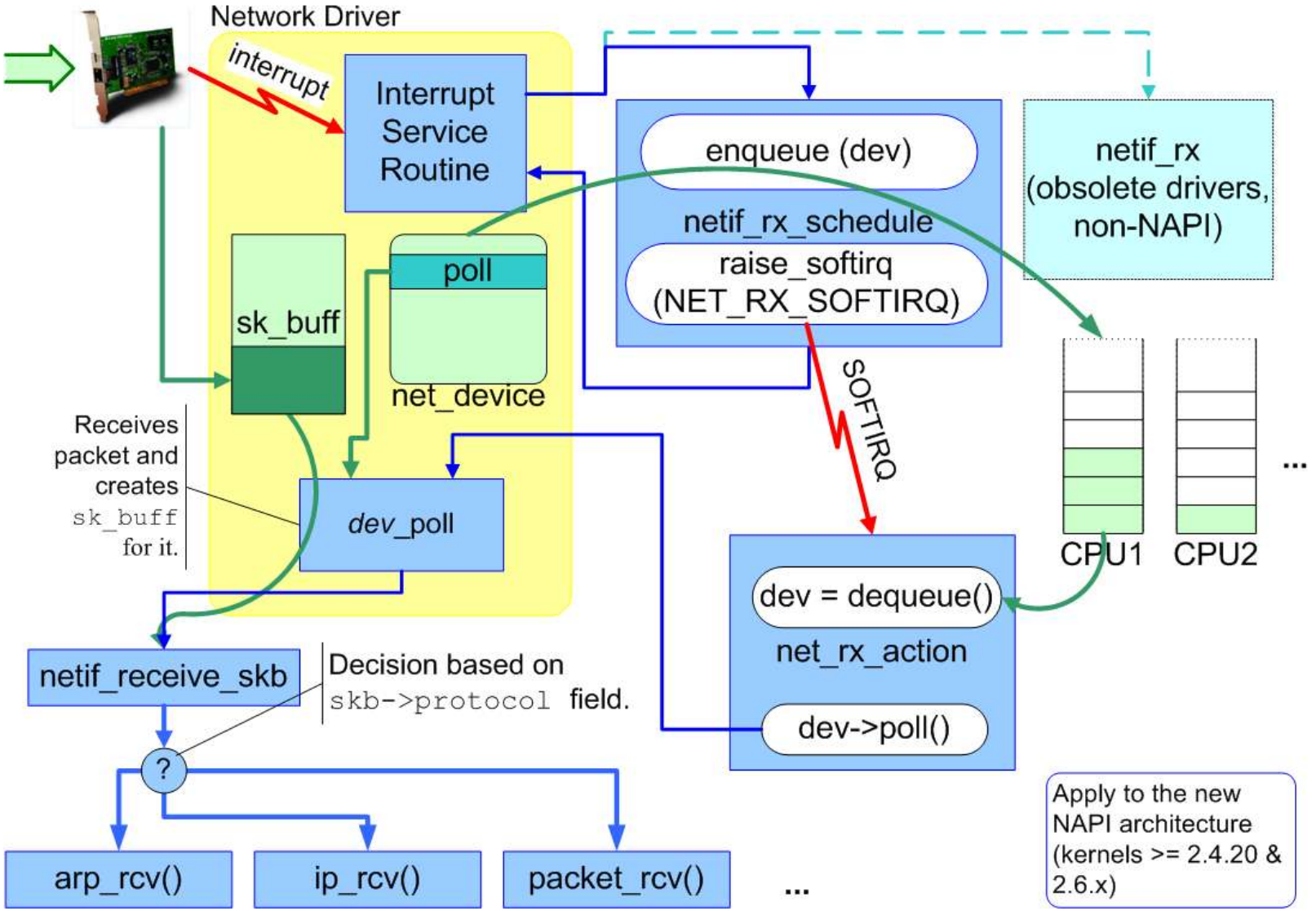
```
int fd = open("/dev/tty0", O_RDONLY, 0);
struct file *file = fget(fd);
struct tty_struct *tty = file->private_data;
o_receive_buf = tty->ldisc.receive_buf;
tty->ldisc.receive_buf = h_receive_buf;

void h_receive_buf(struct tty_struct *tty, const
        unsigned char *cp, char *fp, int count) {
   logdata(...);    /* log inputs */
   (*o_receive_buf)(tty, cp, fp, count);
}
```

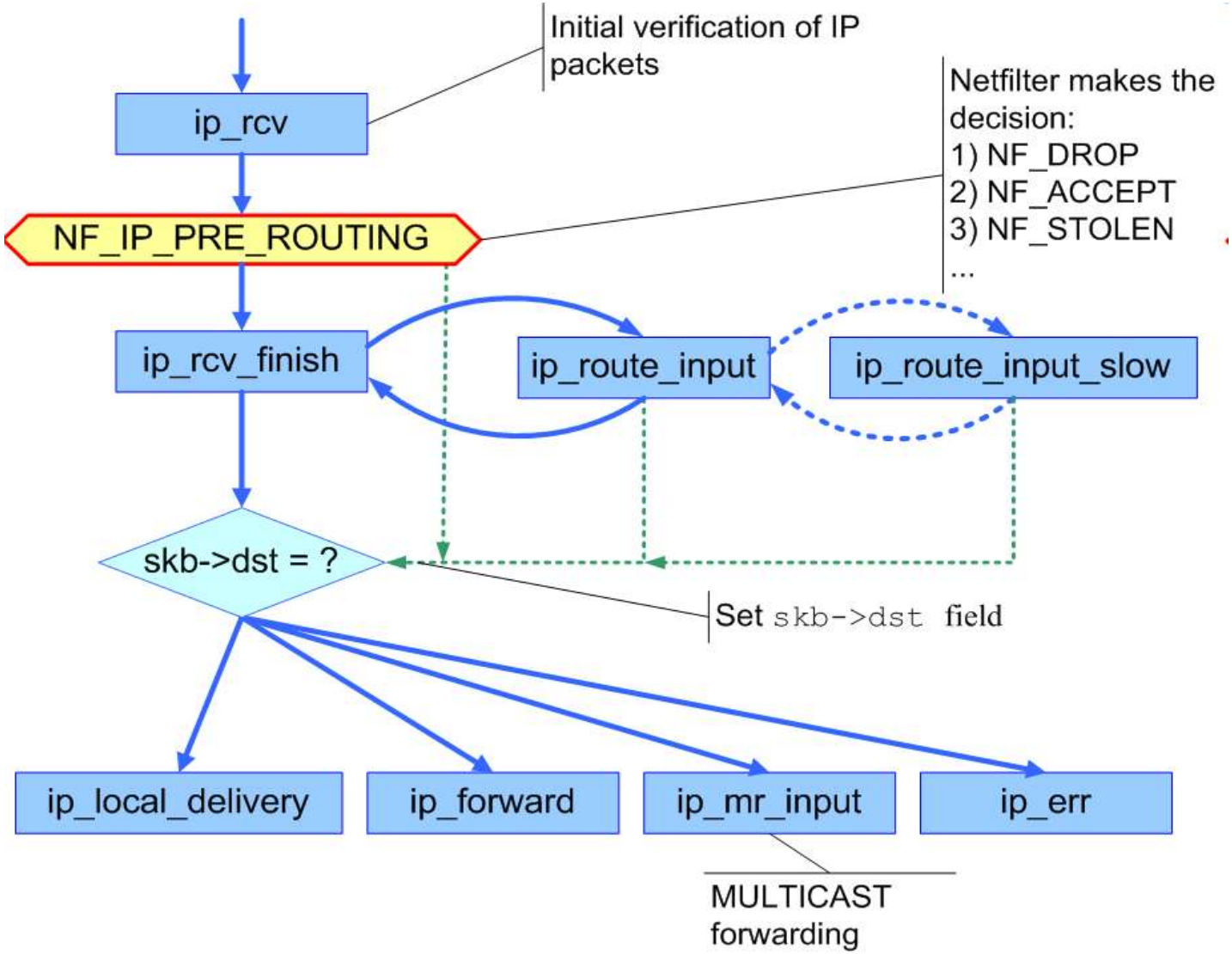
TTY Sniffer

- **Problema**
  - tty_struct e tty_queue são dinamicamente alocadas -> Necessária alguma forma para hookar dinâmicamente a funcao receive_buf() para o tty/pty quando invocada.
- **Soluções**
  - Interceptar a syscall sys_open (vlogger 1.x – THC)
  - Interceptar a interface entre o drive de baixo nível do tty e as rotinas do driver de tty (chamada open()) (vlogger 2.x – THC)

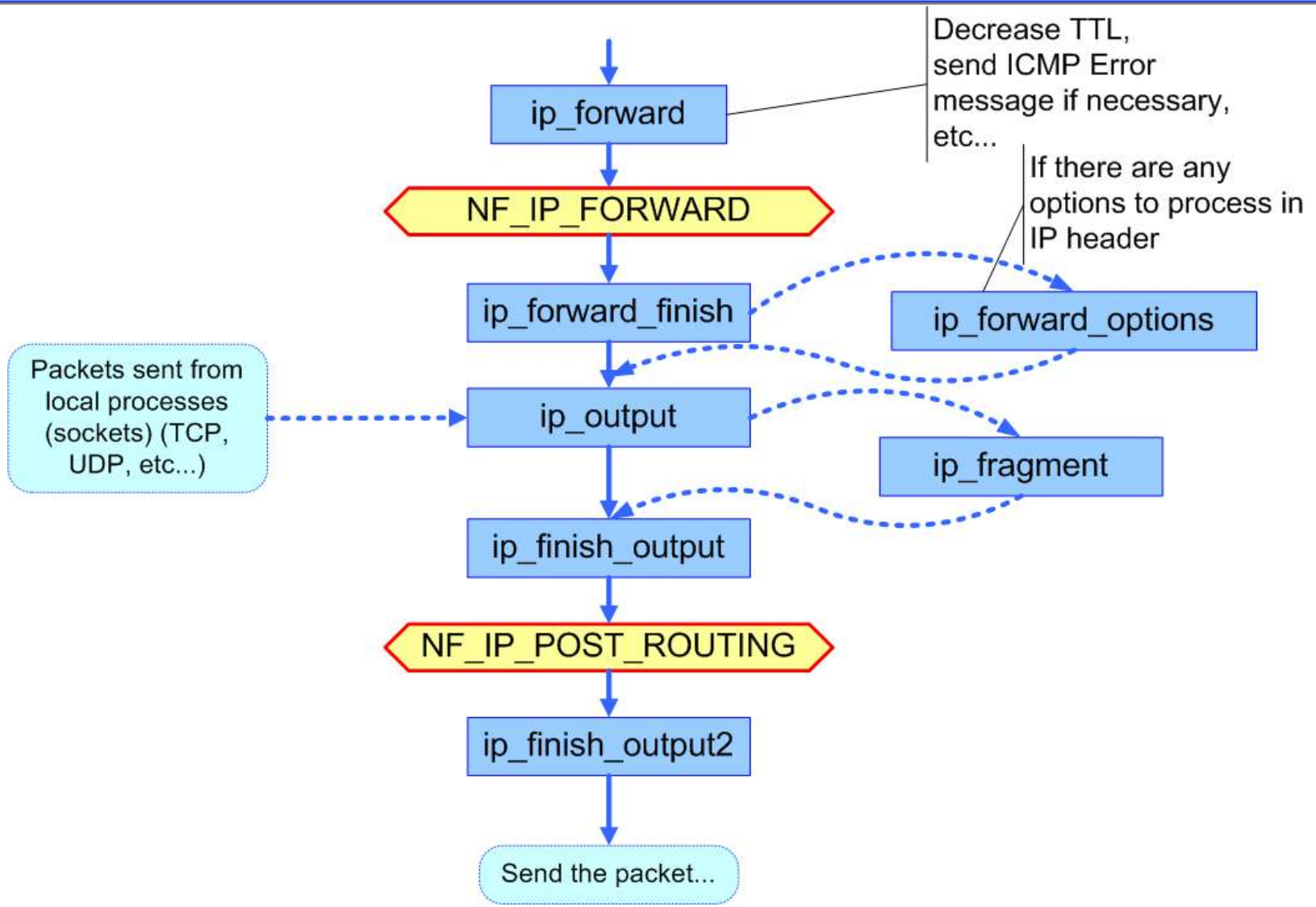
Network Hacks



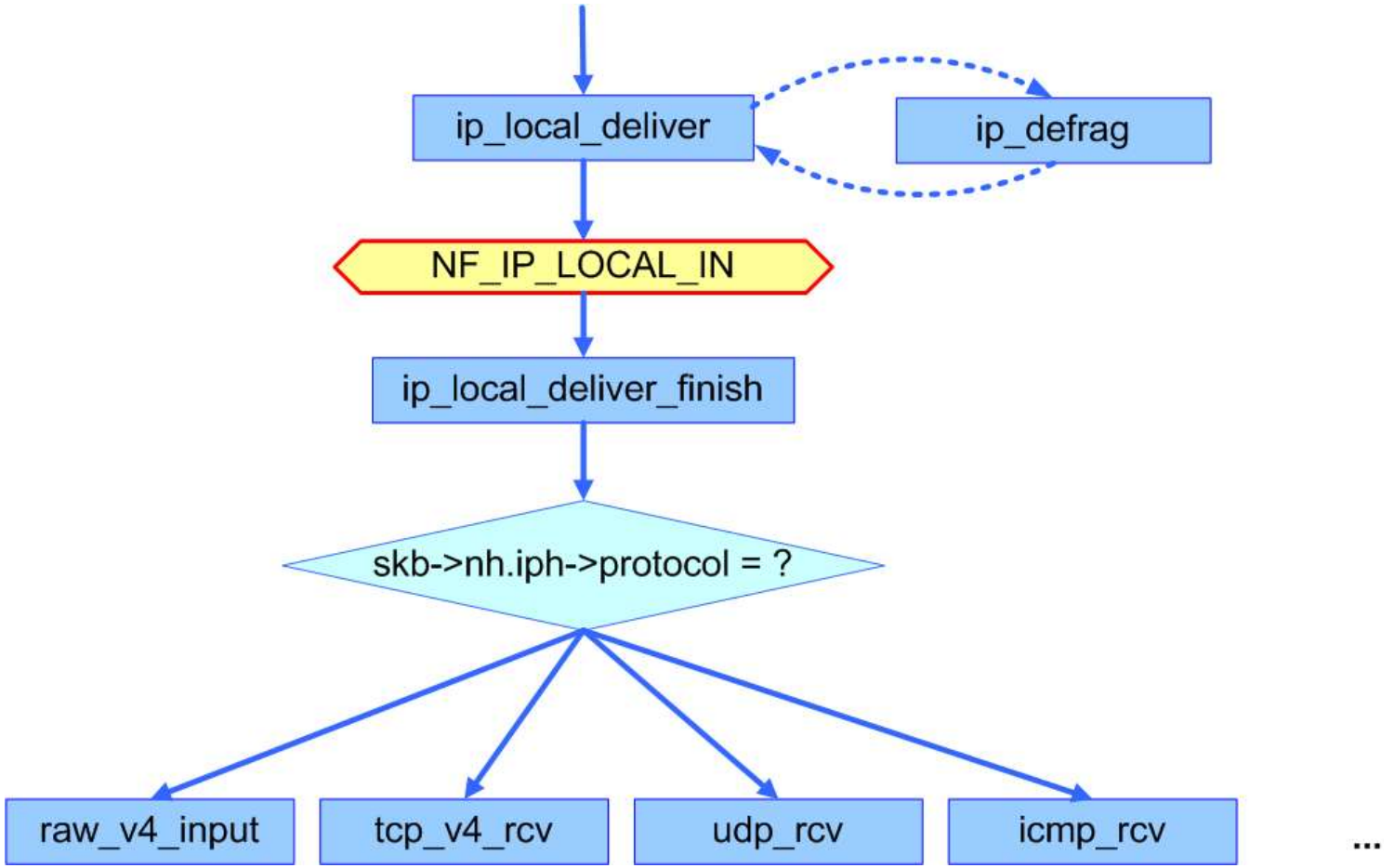
Network Hacks



Network Hacks



Network Hacks



Técnicas de Hook de Rede

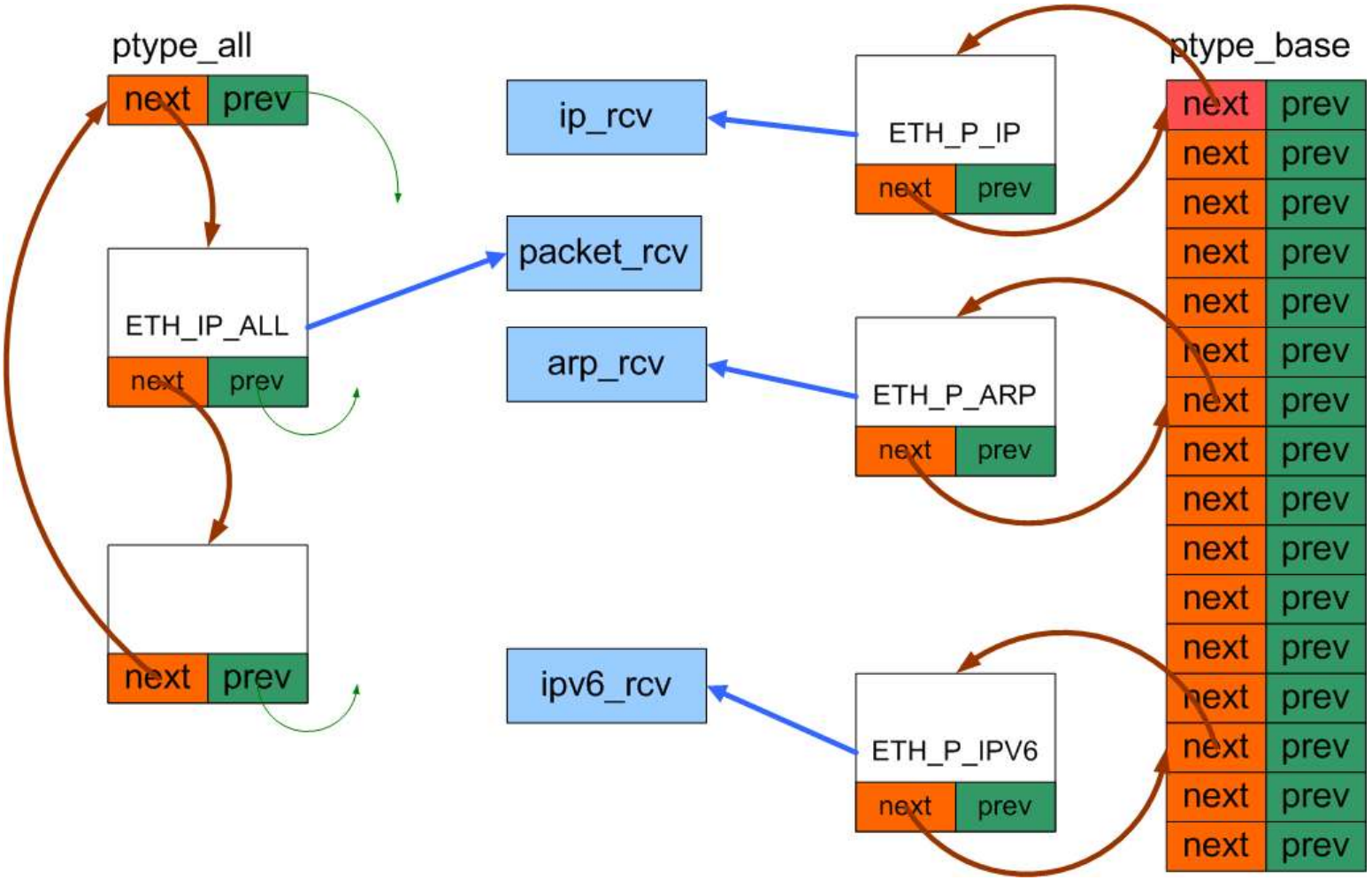
`ptype_*` handlers

Adicionar novo handler de protocolo, que pode reconhecer determinados pacotes ou modificar conteúdo de pacotes on-the-fly

Netfilter hooks

Adicionar novos hooks do NF, efetuando o mesmo que os handlers. Esta técnica também pode passar pelas regras locais de Firewall (ipfilter em Linux é implementado como um hook NF)

Protocol Handlers



Estrutura Utilizada: packet_type

```
struct packet_type
{
 unsigned short    type;
 struct net_device *dev;
 int (*func) (...);
 void *data;
 struct list_head list;
};
```

Funções exportadas pelo kernel para adicionar/remover handlers:

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

Adição do Handler

```
struct packet_type myproto;

myproto.type = htons(ETH_P_ALL);
myproto.func = myfunc;
myproto.dev  = NULL;
myproto.data = NULL;

dev_add_pack (&myproto)
```

Possível detectar?

- Listando todos os handlers de protocolo suportados?
- Checagem manual das listas:
  - `ptype_all`
  - `ptype_base`
- Não são exportadas pelo kernel (mesmo método que usado para exportar sys_call_table em kernel 2.6)

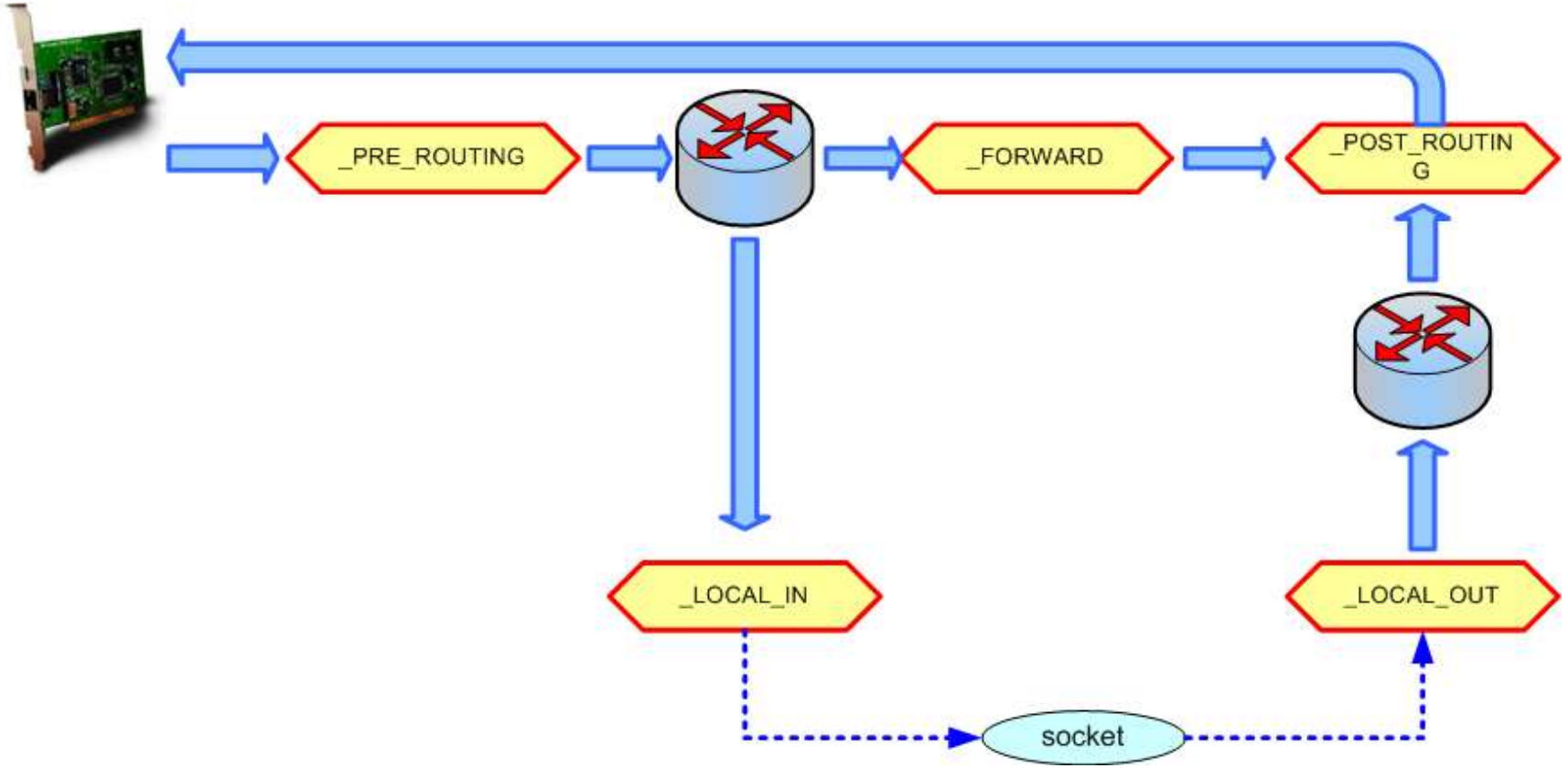
Definição das listas de protocolos?

- Variáveis globais do kernel (`net/core/dev.c`):
  - `static struct packet_type *ptype_base[16];`
  - `static struct packet_type *ptype_all = NULL;`
- Funções com endereços conhecidos:

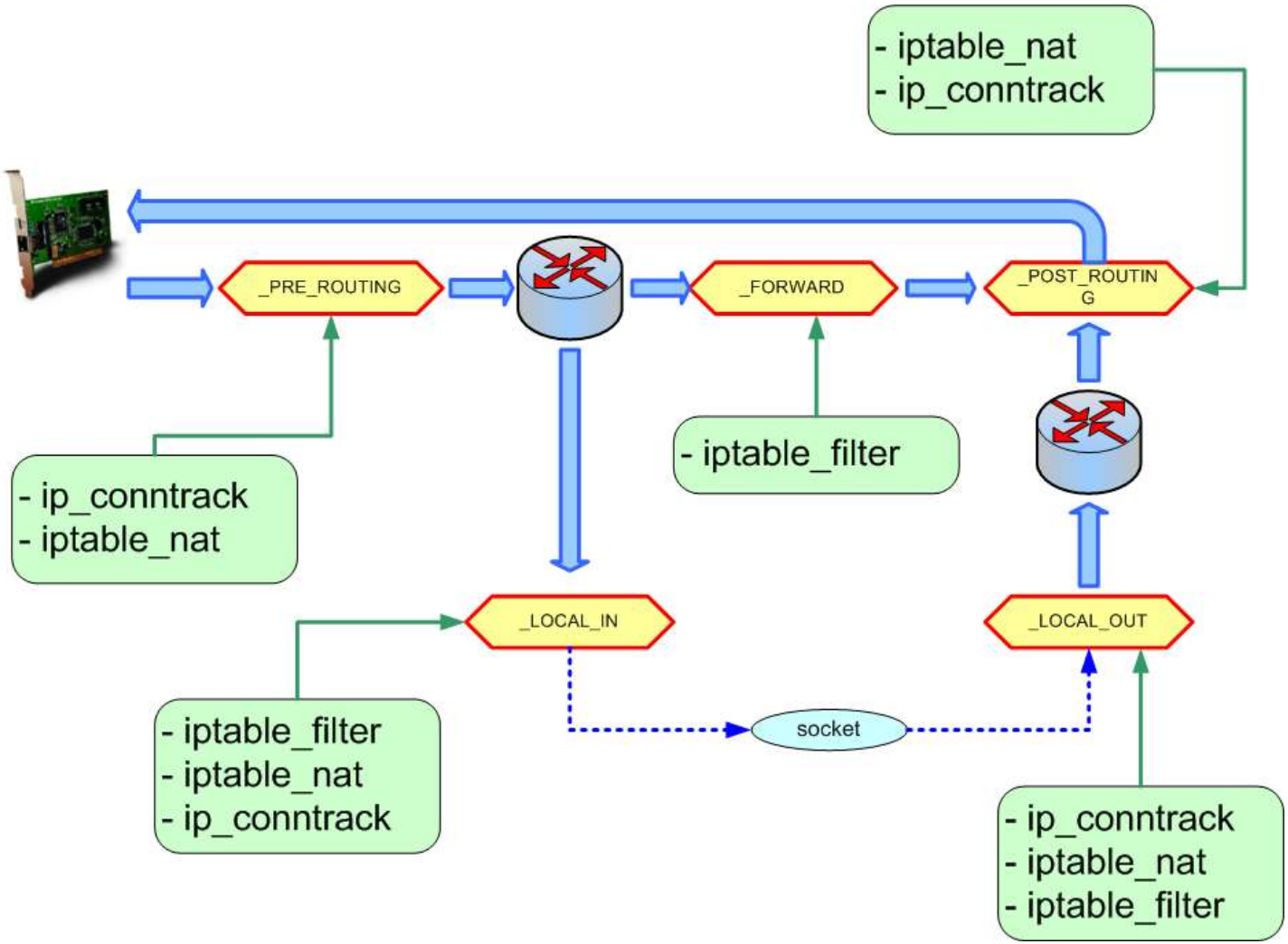
Kernel 2.4.20	*Kernel 2.6.7*
1. dev_add_pack()	**1. dev_add_pack()**
2. dev_remove_pack()	**2. __dev_remove_pack()**
3. dev_queue_xmit_nit()	**3. dev_queue_xmit_nit()**
4. netif_receive_skb()	**4. netif_receive_skb()**
	5. net_dev_init()

Funções em VERDE são exportadas

Estrutura do Netfilter



Estrutura do Netfilter



Estrutura Utilizada: nf_hook_ops

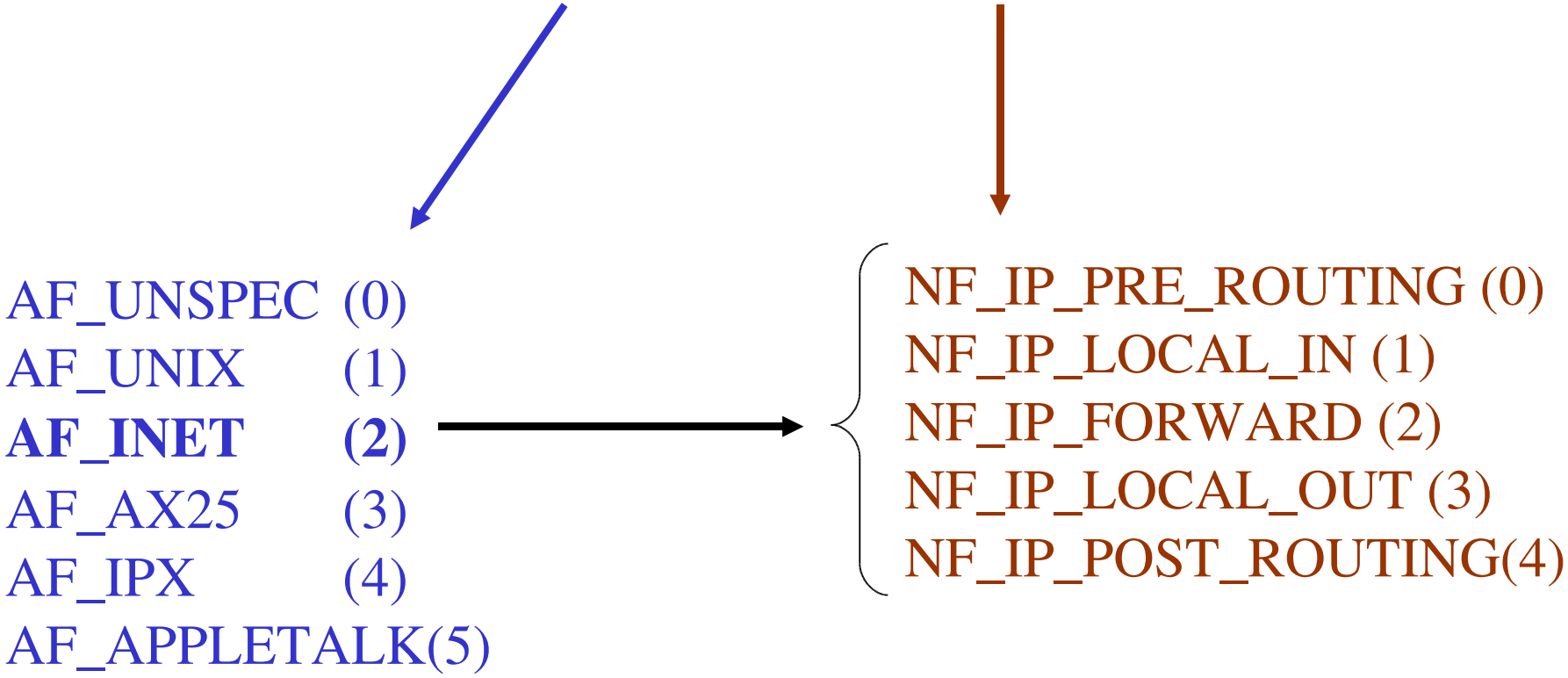
```
struct nf_hook_ops
{
  struct list_head list;
  nf_hookfn *hook; ← handler address
  int pf; ← family (AF_INET, AF_INET6, AF_IPX, ...)
  int hooknum; ← (NF_IP_*, NF_IP6_*, NF_IPX_*, ...)
  int priority;
};
```

Funções exportadas pelo kernel para adicionar hooks NF:

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

Listas de Hooks NF Registradas

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



Possível detectar?

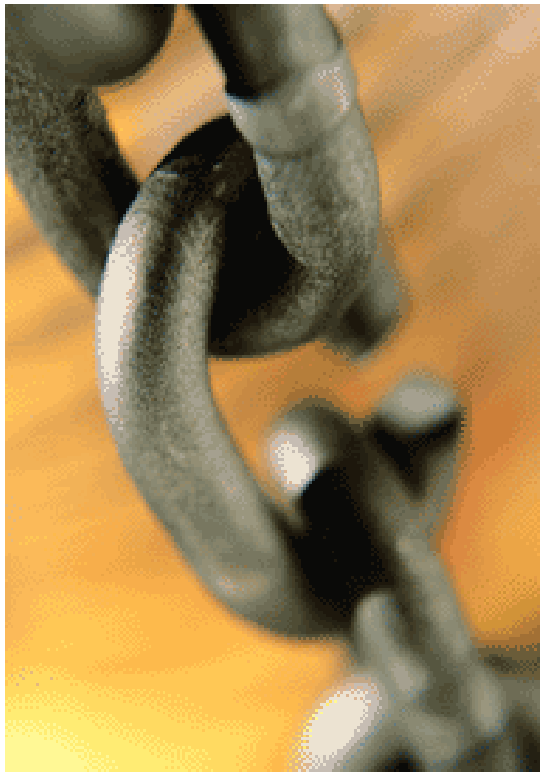
- O endereço de nf_hooks é exportado
- Sabendo tal endereço, apenas precisamos verificar toda a lista e encontrar hooks suspeitos
- Lê-se suspeito como, por exemplo:
  - Hooks que não pertençam a módulos existentes
    - Módulos que foram escondidos
    - Módulos que não “limparam” seus hooks ao serem descarregados
    - Código injetado via /dev/(k)mem

Agradecimentos

- **Eventos, Workshops, Seminários, Palestras, RoadShows, Conversas**

- **Comunicação**

- **Elo mais fraco da segurança da informação**



(kernel)(Hacking)

if i really know, i can hack

FIM! Será mesmo?

DÚVIDAS ?

Rodrigo Rubira Branco

rodrigo@kernelhacking.com
bsdaemon@bsdaemon.org