

# TÉCNICAS DE OCULTAÇÃO DE TRÁFEGO DE REDE EM HONEYPOTS DE ALTA INTERATIVIDADE

**Luiz Gustavo C. Barbato**

Laboratório Associado de Computação e Matemática Aplicada - LAC  
Instituto Nacional de Pesquisas Espaciais - INPE/MCT  
Av. dos Astronautas, 1758 – 12227-010 São José dos Campos, SP  
lgbarbato@lac.inpe.br

**Antonio Montes**

Centro de Pesquisas Renato Archer - CenPRA/MCT  
Rodovia Dom Pedro I, km 143,6 – 13082-120 Campinas, SP  
antonio.montes@cenpra.gov.br

## RESUMO

*A captura das atividades de atacantes e a transmissão dessas informações de honeypots para a estação de monitoração, não é um processo trivial e requer algumas precauções para evitar denunciar a honeynet. Neste artigo, técnicas para esconder este tráfego de rede são discutidas. Resultados do emprego de uma dessas técnicas na ferramenta de monitoração SMaRT também são apresentados.*

## ABSTRACT

*The recording of attackers' activities and the transmission of this information from honeypots to monitoring stations is not a trivial process and requires some precautions in order to avoid given away the honeynet. In this paper, techniques used to hide this network traffic from the attacker are discussed. Results from one of these techniques used with the SMaRT monitoring tool are also presented.*

## 1 INTRODUÇÃO

Monitorar atividades dos atacantes em *honeypots* de alta interatividade [1] é um dos métodos utilizados para determinar quais vulnerabilidades estão sendo exploradas para ganhar acesso às máquinas, capturar novas ferramentas de ataque e entender a motivação que os levaram a iniciar os ataques [2]. Um dos maiores problemas deste processo é capturar estas atividades e transferir estes dados de forma indetectável pelos atacantes.

A captura destas atividades é feita por meio de técnicas de monitoração em *kernel space* implementadas através de um módulo de *kernel* [3]. Este mesmo módulo de *kernel* é utilizado para transmitir os dados capturados e também pode ser utilizado para esconder a transmissão destas informações dos atacantes que agem nos *honeypots*.

Como é comum a utilização de monitores de tráfego de rede, em máquinas comprometidas, em busca de senhas e dados confidenciais, o mesmo ocorre em *honeynets*. As formas de invadir e usar essas máquinas são as mesmas, visto que o atacante não percebe que está em uma *honeynet*. Mecanismos devem ser implementados nos *honeypots* para iludir o atacante e não deixar que ele note suas ações sendo exportadas pela rede.

Uma alternativa para isso seria o uso de criptografia. Entretanto, a utilização de criptografia na transação não resolve por completo este problema. Esta técnica apenas esconderia o conteúdo dos pacotes, porém estes mesmos continuam sendo vistos trafegando pela rede. Os dados devem ser coletados porém não podem ser descobertos e a transparência de todo o processo indica o sucesso da monitoração.

Neste artigo serão mostradas algumas técnicas que permitem ocultar o tráfego de dados pela rede, iludindo aplicações como os monitores de tráfego de rede. É importante destacar que estas técnicas iludem apenas os atacantes que estão agindo nos *honeypots* onde elas foram implantadas, pois estas são baseadas na modificação de partes centrais do sistema operacional<sup>1</sup>, responsáveis por tratar o fluxo de dados desde a captura dos pacotes pela placa de rede até o seu recebimento pelos usuários.

E como resultados da implantação destas técnicas, será mostrado uma sessão de teste capturada com o sistema SMaRT (Session Monitoring and Replay Tool) [4] desenvolvido especialmente para este propósito.

## 2 FLUXO DE DADOS NO KERNEL

O processo normal de recepção de dados da rede se inicia quando a placa de rede detecta dados, na forma de quadros, passando pelo barramento. Neste momento a placa verifica se os quadros são destinados a ela e em caso afirmativo repassa-os para o *kernel*. Os quadros que não forem destinados a ela são rejeitados. O *kernel* ao receber estes dados verifica o tipo de protocolo que os quadros carregam e os coloca em uma fila específica. Em seguida uma rotina específica de tratamento retira os dados da fila e verifica se existe algum processo de usuário que esteja esperando por eles. Caso haja, eles são repassados para este, caso contrário uma mensagem de erro é enviada para o remetente.

<sup>1</sup>O sistema operacional em questão, para o qual as técnicas foram desenvolvidas é o Linux versão 2.4

Uma outra forma de recepção é quando a placa de rede está configurada para trabalhar em modo promíscuo. Esta configuração permite que a placa capture todos os quadros que trafegam pelo barramento, independentemente de protocolo e ignorando a condição destes dados serem destinados a ela. O *kernel* então recebe estes dados e os coloca em uma fila de protocolos genéricos<sup>2</sup> para depois disponibilizar os dados para os usuários.

Nesta seção será discutido de forma detalhada, o fluxo de dados de rede desde a captura pela placa<sup>3</sup> até os dados serem entregues para as funções tratadoras dos protocolos de rede, para melhor entendimento das técnicas que serão apresentadas (Figura 1). O fluxo de dados foi baseado em placas de rede que utilizam o *driver* 8390.o (*drivers/net/8390.c*)<sup>4</sup>, pois a maioria dos *honeypots* da *honeynet* utilizam este *driver*, e é este que será tomado como base durante a discussão. Por fim, será discutido também como o Linux registra os *sockets* do tipo PF\_PACKET [5] para permitir que as aplicações recebam todos os dados dos protocolos genéricos.

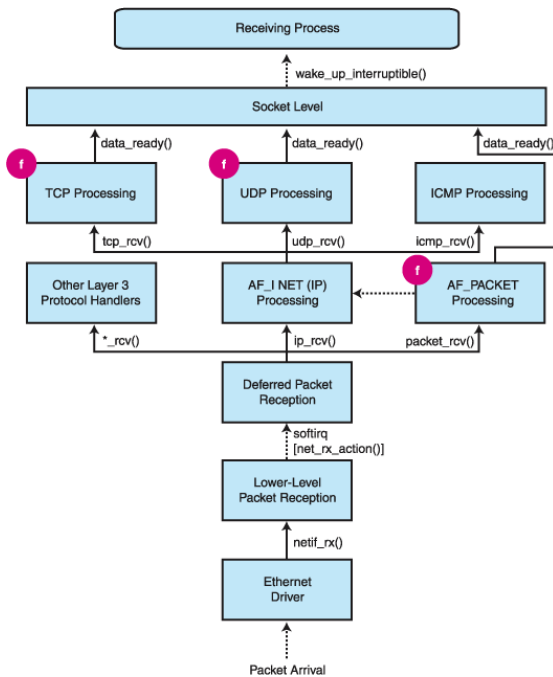


Figura 1: Caminho da recepção dos pacotes e as funções mais importantes  
<http://www.linuxjournal.com/modules/NS-1j-issues/issue95/5617f1.png>

Para melhor entendimento do artigo, abaixo será mostrado um diagrama de encadeamento entre as fun-

<sup>2</sup>Neste artigo o termo “protocolos genéricos” será entendido como protocolos de todos os tipos e “pacotes genéricos” como sendo pacotes de protocolos genéricos

<sup>3</sup>Durante a discussão, supõe-se que a placa de rede esteja configurada para trabalhar em modo promíscuo.

<sup>4</sup>Os diretórios que serão descritos neste artigo encontram-se no código fonte do Linux, geralmente em /usr/src/linux

ções que serão utilizadas.

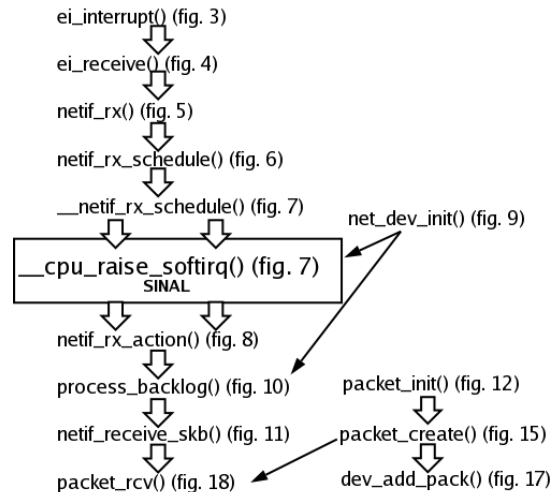


Figura 2: Diagrama de encadeamento entre as funções

### 2.1 Visão detalhada

A placa de rede ao detectar algum quadro trafegando pelo barramento, o captura e o copia para sua memória. Ao terminar a recepção de todo o quadro, a placa gera um pedido de interrupção sinalizando que está pronta para repassar os dados. Neste momento o *driver* da placa de rede é acionado para tratar a interrupção (*ei\_interrupt*) gerada e iniciar o processo de recepção (*ei\_receive*) (Figura 3).

```
void ei_interrupt(int irq, void *dev_id,
                 struct pt_regs *regs)
{
    struct net_device *dev = dev_id;
    ...
    ei_receive(dev);
    ...
}
```

Figura 3: Função responsável por tratar a interrupção gerada pela placa de rede

O *driver* então, aloca (*dev\_alloc\_skb*) uma estrutura denominada *sk\_buff* (*include/linux/skbuff.h*), que é a representação dos dados perante o *kernel*, extrai os dados da placa de rede e os coloca (*ei\_block\_input*) nesta estrutura, para finalmente invocar a rotina (*netif\_rx*) responsável por transferi-los para estruturas internas do *kernel*<sup>5</sup> (Figura 4). Essa rotina é denominada gerenciadora de recepção de pacotes genéricos, pois até este ponto nenhum protocolo foi analisado ainda.

<sup>5</sup>Importante destacar que os *drivers* fazem parte do *kernel* pois são módulos que agregam funcionalidades ao mesmo, mas para simplificar a explicação durante o decorrer deste artigo, a comunicação entre o *driver* e as estruturas internas do *kernel* serão tratadas como se os *drivers* não fizessem parte do *kernel*

```

static void ei_receive(struct net_device *dev)
{
    ...
    struct sk_buff *skb;

    skb = dev_alloc_skb(pkt_len+2);
    ...
    ei_block_input(dev, pkt_len, skb, current_offset + \
                  sizeof(rx_frame));

    ...
    netif_rx(skb);
    ...
}

```

Figura 4: Função responsável por extrair os dados da placa de rede

Já o *kernel*, ao receber (*netif\_rx*) (*net/core/dev.c*) os dados, inicia o processo de enfileiramento (*\_\_skb\_queue\_tail*) (*include/linux/skbuff.h*) verificando qual CPU (*smp\_processor\_id()*) está disponível no momento, em caso de SMP, para tratar a fila. Logo em seguida é iniciado o processo de sinalização (*netif\_rx\_schedule*) (*include/linux/netdevice.h*) para indicar que os dados estão prontos para serem manipulados (Figura 5).

```

int netif_rx(struct sk_buff *skb)
{
    int this_cpu = smp_processor_id();
    struct softnet_data *queue;
    ...
    queue = &softnet_data[this_cpu];
    ...
    __skb_queue_tail(&queue->input_pkt_queue, skb);
    ...
    netif_rx_schedule(&queue->blog_dev)
    ...
}

```

Figura 5: Função gerenciadora de recepção de pacotes genéricos

Após o processo de enfileiramento, é gerada (*\_\_cpu\_raise\_softirq*) (*include/linux/interrupt.h*) uma interrupção para indicar que os dados foram recebidos com sucesso e estão prontos para serem tratados (Figura 6, 7).

```

... void netif_rx_schedule(struct net_device *dev)
{
    if (netif_rx_schedule_prep(dev))
        __netif_rx_schedule(dev);
}

```

Figura 6: Chamada da função de sinalização da recepção de dados

```

... void __netif_rx_schedule(struct net_device *dev)
{
    ...
    int cpu = smp_processor_id();
    ...
    __cpu_raise_softirq(cpu, NET_RX_SOFTIRQ);
    ...
}

```

Figura 7: Geração de interrupção sinalizando a recepção de dados

Ao receber esta nova interrupção, o *kernel* invoca a função responsável por tratar o sinal *NET\_RX\_SOFTIRQ* denominada *net\_rx\_action* (*/net/core/dev.c*) (Figura 8). Esta função verifica (*list\_entry*) qual a interface de rede (*dev*) foi a responsável pela recepção dos dados e invoca a função que tem a finalidade de desenfileirá-los (*poll*) (Figura 9).

```

static void net_rx_action(struct softirq_action *h)
{
    int this_cpu = smp_processor_id();
    struct softnet_data *queue = &softnet_data[this_cpu];
    ...
    dev = list_entry(queue->poll_list.next,
                   struct net_device, poll_list);
    ...
    dev->poll(dev, &budget)
    ...
}

```

Figura 8: Função responsável por tratar a interrupção *NET\_RX\_SOFTIRQ*

A função *poll* (Figura 9) é um ponteiro para a função *process\_backlog* (Figura 10), associada no momento da inicialização (*net\_dev\_init*) (*net/core/dev.c*) da fila de pacotes de entrada (*softnet\_data*). Nesta mesma função (*open\_softirq*) também é associado o sinal de interrupção *NET\_RX\_SOFTIRQ* à função *net\_rx\_action*, ambos citados anteriormente (Figura 9).

```

int __init net_dev_init(void)
{
    ...
    /* Initialise the packet receive queues. */

    for (i = 0; i < NR_CPUS; i++) {
        struct softnet_data *queue;

        queue = &softnet_data[i]
        ...
        queue->blog_dev.poll = process_backlog;
        ...
    }
    ...
    open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
    ...
}

```

Figura 9: Inicialização da filas de recepção de pacotes

A função *process\_backlog* então retira um a um os dados da fila e chama a rotina responsável por tratar

a recepção destes, nomeada *netif\_receive\_skb* (`net/core/dev.c`) (Figura 10).

```
static int process_backlog(struct net_device *blog_dev,
                          int *budget)
{
    ...
    for (;;) {
        struct sk_buff *skb;
        ...
        skb = __skb_dequeue(&queue->input_pkt_queue);
        ...
        netif_receive_skb(skb);
        ...
    }
    ...
}
```

Figura 10: Retirada dos pacotes da fila

É na função *netif\_receive\_skb* que é feito o trabalho principal de analisar os dados recebidos e encaminhá-los para as duas listas denominadas *ptype\_all* (protocolos genéricos) e *ptype\_base* (protocolos específicos) (Figura 11). Isto feito, é invocada a função para iniciar a recepção do pacote na camada de rede, que, no caso do protocolo IP é a função *ip\_rcv* (*func*) (`net/ipv4/ip_input.c`) de acordo com a inicialização e o registro deste protocolo [6]. Já a função que trata o recebimento de pacotes genéricos é a *packet\_rcv* (*func*) (`net/packet/af_packet.c`) (Figura 18) como será mostrado a seguir.

```
int netif_receive_skb(struct sk_buff *skb)
{
    struct packet_type *ptype, *pt_prev;
    ...
    unsigned short type = skb->protocol;
    ...
    for (ptype = ptype_all; ptype; ptype = ptype->next) {
        ...
        ret = pt_prev->func(skb, skb->dev, pt_prev);
        ...
    }
    ...
    for (ptype=ptype_base[ntohs(type)&15]; ptype; \
         ptype=ptype->next) {
        ...
        ret = pt_prev->func(skb, skb->dev, pt_prev);
        ...
    }
    ...
}
```

Figura 11: Filas de protocolos genéricos e específicos

E, por fim, cada camada de protocolo exerce sua função específica, passando os dados camada a camada até chegarem aos processos de aplicação [6].

## 2.2 Registro de drivers de protocolos genéricos

Os *drivers*<sup>6</sup> de protocolo implementam as várias arquiteturas de protocolos disponíveis na máquina, como TCP/IP, IPX/SPX e NetBIOS. Eles definem

<sup>6</sup>Estes *drivers* podem ser implementados através de módulos de *kernel*.

uma interface usada pelas aplicações para troca de dados. Além destes protocolos específicos disponíveis, existe uma outra implementação de *driver* para tratar protocolos genéricos [5]. Aplicações desenvolvidas para trabalharem com protocolos genéricos podem capturar qualquer dado que trafega pela rede independentemente destes serem destinados a ela ou não e do tipo de protocolo, que é o caso dos monitores de tráfego de rede.

Neste caso, o módulo de *kernel* responsável por esta implementação é o *af\_packet* (`net/packet/af_packet.c`). A função *packet\_init*, chamada toda vez que o módulo for carregado, foi projetada para registrar as operações que podem ser feitas com os *sockets* do tipo *PF\_PACKET* [5]. Após o carregamento do módulo é invocada a função *sock\_register* (`net/socket.c`) que registrará a função responsável pela criação destes tipos de *sockets* (Figura 12).

```
static int __init packet_init(void)
{
    sock_register(&packet_family_ops);
    ...
}
```

Figura 12: Função de inicialização do módulo *af\_packet*

A estrutura *packet\_family\_ops* é do tipo *net\_proto\_family* (`include/linux/net.h`) e possui as seguintes entradas (Figura 13):

```
static struct net_proto_family packet_family_ops = {
    family:      PF_PACKET,
    create:      packet_create,
};
```

Figura 13: Associação da função de criação de *sockets* *PF\_PACKET*

Esta estrutura indica que a função *packet\_create* (Figura 15) será invocada toda vez que for criado um *socket* do tipo *PF\_PACKET* [5] (Figura 13). Internamente, esta função associa as operações deste *socket* à estrutura *packet\_ops* (Figura 14), dentre elas a responsável por receber mensagens (*packet\_recvmsg*).

```
struct proto_ops packet_ops = {
    family:      PF_PACKET,
    ...
    recvmsg:    packet_recvmsg
    ...
}
```

Figura 14: Operações que podem ser feitas com *sockets* *PF\_PACKET*

```

... int packet_create(struct socket *sock, int protocol)
{
    struct sock *sk;
    ...
    sock->ops = &packet_ops;
    ...
    sk->protinfo.af_packet->prot_hook.func = packet_rcv;
    ...
    dev_add_pack (&sk->protinfo.af_packet->prot_hook);
    ...
}

```

Figura 15: Função de criação dos *sockets PF\_PACKET*

O ponteiro *prot\_hook* é uma estrutura do tipo *packet\_type* (`include/linux/netdevice.h`) (Figura 16), que associa o tipo de pacote *ETH\_P\_ALL* à função *packet\_rcv* (`net/packet/af_packet.c`) (Figura 18), responsável por tratar o recebimento de dados de todos os protocolos. Assim, toda vez que qualquer pacote for recebido a função *packet\_rcv* será invocada.

Importante destacar que a função *packet\_rcv* é associada a um dos itens da estrutura *packet\_type* (Figura 16), um ponteiro de função nomeado *func* (Figura 15). Este ponteiro *func* foi invocado pela função *netif\_receive\_skb* logo após o desenfileiramento dos dados no fluxo de dados discutido conforme mostrado na Figura 11.

```

struct packet_type
{
    ...
    int (*func) (struct sk_buff *, struct net_device *, \
                struct packet_type *);
    ...
};

```

Figura 16: Definição da estrutura *packet\_type*

Uma outra atividade desta mesma função que é importante mencionar é a chamada da função *dev\_add\_pack* (`net/core/dev.c`) (Figura 17). Basicamente, esta função insere pacotes do tipo *ETH\_P\_ALL* na lista *ptype\_all*, a mesma mostrada na Figura 11.

```

void dev_add_pack(struct packet_type *pt)
{
    ...
    if (pt->type == htons(ETH_P_ALL)) {
        ...
        pt->next=ptype_all;
        ptype_all=pt;
    }
    ...
}

```

Figura 17: Registro do protocolo genérico

A função *packet\_rcv*, em determinado ponto, verifica (*sk\_run\_filter*) (`net/core/filter.c`) os filtros BPF [7] com intuito de entregar os dados para aplicações que utilizam a biblioteca *pcap*, ou rejeitar os mesmos, caso os filtros assim determinem (Figura 18).

Passando pelos filtros, os dados são inseridos (`__skb_queue_tail`) (`include/linux/skbuff.h`) em uma outra fila e as aplicações dos usuários são sinalizadas (*data\_ready*) informando que os dados já estão prontos para serem analisados (Figura 18). Neste ponto os monitores de tráfego de rede estão aptos a mostrarem os dados para os usuários.

```

static int packet_rcv(struct sk_buff *skb,
                    struct net_device *dev,
                    struct packet_type *pt)
{
    ...
    res = sk_run_filter(skb, sk->filter->insns, \
                      sk->filter->len);
    ...
    __skb_queue_tail(&sk->receive_queue, skb);
    ...
    sk->data_ready(sk, skb->len);
}

```

Figura 18: Função de recepção de protocolos genéricos

### 2.3 Representação dos pacotes perante o kernel

Desde a placa de rede até antes de serem entregues aos usuários, os pacotes caminham dentro do *kernel* representados por estruturas do tipo *sk\_buff*.

A seguir serão mostradas as principais entradas da estrutura *sk\_buff*<sup>7</sup> que serão importantes no decorrer deste artigo. Essas entradas se referem a alguns protocolos que esta estrutura pode manipular.

#### 2.3.1 Camada de enlace

Nesta seção é mostrado como manipular os dados do protocolo *Ethernet*:

```

/* Link layer header */
union
{
    struct ethhdr *ethernet;
    unsigned char *raw;
} mac;

```

Figura 19: Estruturas da camada de enlace

Os dados do protocolo *Ethernet* podem ser acessados através da estrutura *mac*.

**Exemplo:** (`include/linux/if_ether.h`)

```

//Endereço de origem do protocolo Ethernet:
source_mac = skb->mac.ethernet->h_source;

//Endereço de destino do protocolo Ethernet:
dest_mac = skb->mac.ethernet->h_dest;

```

Figura 20: Manipulação dos dados do protocolo *Ethernet*

<sup>7</sup>A variável *skb* é do tipo *sk\_buff*

### 2.3.2 Camada de rede

Nesta seção é mostrado como manipular os dados do protocolo *IP*:

```
/* Network layer header */
union
{
    struct iphdr    *iph;
    struct ipv6hdr  *ipv6h;
    struct arphdr   *arph;
    struct ipxhdr   *ipxh;
    unsigned char   *raw;
} nh;
```

Figura 21: Estruturas da camada de rede

Agora, os dados do protocolo *IP* podem ser acessados através da estrutura *nh*.

**Exemplo:** (include/linux/ip.h)

```
//Endereço de origem do protocolo IP:
source_ip = skb->nh.iph->saddr;

//Endereço de destino do protocolo IP:
dest_ip = skb->nh.iph->daddr;
```

Figura 22: Manipulação dos dados do protocolo *IP*

### 2.3.3 Camada de transporte

Nesta seção é mostrado como manipular os dados do protocolo *UDP* e *TCP*:

```
/* Transport layer header */
union
{
    struct tcphdr   *th;
    struct udphdr   *uh;
    struct icmp_hdr *icmph;
    struct igmp_hdr *igmp;
    struct iphdr    *iph;
    struct spxhdr   *spx;
    unsigned char   *raw;
} h;
```

Figura 23: Estruturas da camada de transporte

Já os dados dos protocolos *UDP* e *TCP* podem ser acessados através da estrutura *h*.

**Exemplo 1:** (include/linux/udp.h)

```
//Número de porta de origem do protocolo UDP:
source_port = skb->h.uh->source;

//Número de porta de destino do protocolo UDP:
dest_port = skb->h.uh->dest;
```

Figura 24: Manipulação dos dados do protocolo *UDP*

**Exemplo 2:** (include/linux/tcp.h)

```
//Número de porta de origem do protocolo TCP:
source_port = skb->h.th->source;

//Número de porta de destino do protocolo TCP:
dest_port = skb->h.th->dest;
```

Figura 25: Manipulação dos dados do protocolo *TCP*

### 2.3.4 Camada de aplicação

Nesta seção é mostrado como manipular os dados das aplicações:

```
/* Data head pointer */
unsigned char *data;
```

Figura 26: Variável de representação dos dados da camada de aplicação

E por fim, os dados que são transmitidos pelas aplicações podem ser acessados diretamente através do ponteiro *data*.

**Exemplo:**

```
//Área de dados
data = skb->data;
```

Figura 27: Manipulação dos dados da camada de aplicação

## 3 TÉCNICAS DE OCULTAÇÃO DE TRÁFEGO DE REDE

Depois de mostrada uma parte do fluxo de dados (Figura 1), é possível discutir os locais mais apropriados para ocultar o tráfego de rede. É importante destacar que os filtros de tráfego de rede somente funcionarão para ocultar a ação do atacante dos *honeypots* do mesmo barramento que possuem a técnica aplicada, visto que a forma de transmissão dos dados já automaticamente impede que as aplicações do *honeypot* os capturem, porque eles são inseridos diretamente no *buffer* da placa de rede, ignorando assim, toda a implementação da pilha de rede (TCP/IP) do *honeypot*<sup>8</sup>.

Assim como as técnicas em *kernel space* de monitoração de atividades [3], as técnicas de ocultação de tráfego também podem ser implementadas através de módulos de *kernel* [8] ou modificações direto no código fonte do mesmo. Esses filtros são baseados na verificação de certos campos dos cabeçalhos dos protocolos, como mostrados na seção anterior, previamente definidos.

<sup>8</sup>A forma de transferência dos dados também pode ser considerada uma técnica de ocultar tráfego de rede, porém somente aplicada ao *honeypot* que está transmitindo as informações.

Os filtros podem ser colocados tanto diretamente nos *drivers* das placas de rede, na função gerenciadora de recepção de pacotes genéricos ou através da modificação do módulo de *kernel* [8] de tratamento de protocolos genéricos. Estes filtros também podem ser aplicados em outros locais, dentro do fluxo de dados apresentado, desde que sejam colocados antes da sinalização das aplicações dos usuários. A seguir serão mostradas três técnicas/formas diferentes de implementar os filtros mencionados.

### 3.1 Driver da placa de rede

Os *drivers* das placas de rede podem ser previamente desenvolvidos visando ocultar certos tráfegos. Como estes códigos são os responsáveis pela transferência de dados entre o equipamento (placa) e o sistema operacional, os filtros podem ser inseridos antes que os dados sejam repassados para o *kernel*.

De acordo com o fluxo apresentado, o *driver* (*drivers/net/8390.c*) é invocado (*ei\_interrupt*) após a interrupção gerada pela placa de rede. Neste ponto a função de recepção (*ei\_receive*) do *driver* irá arrumar os dados capturados, em uma forma que o *kernel* entenda (*sk\_buff*), e repassá-los (*netif\_rx*) para o mesmo. Um outro ponto importante a ser ressaltado, é a utilização da função de interface entre o *driver* e o *kernel*, a *netif\_rx*. Se a função *netif\_rx* é a responsável por repassar os dados para o *kernel*, é possível inserir um filtro rejeitando-os antes da sua chamada (Figura 28).

```
static void ei_receive(struct net_device *dev)
{
    ...
    if ( skb->mac.ethernet->h_source != hidden_h_source )
        netif_rx(skb);
    ...
}
```

Figura 28: Alteração do *driver* da placa de rede

Neste exemplo foi mostrado como esconder o tráfego de rede de determinado endereço *Ethernet* de origem (*hidden\_h\_source*) dos processos de usuários.

A vantagem de utilizar esta técnica é o fato de dificultar a localização do filtro sendo que não é necessário a criação de nenhum módulo de *kernel* à parte, utilizando somente o módulo da placa de rede.

Em contrapartida, para cada modelo de placa de rede seria necessário a inserção deste filtro no código do respectivo *driver*.

### 3.2 Gerenciador de recepção de pacotes genéricos

Uma outra forma de ocultar o tráfego de rede é modificar diretamente a função *netif\_rx* (*net/core/dev.c*), sendo que esta, como foi dito, é chamada durante a transferência de dados entre o *driver* e o *kernel*. Neste caso o melhor local para colocar o filtro é

no início da função, evitando que certos dados sejam contabilizados nas estatísticas de pacotes recebidos e rejeitados (Figura 29).

```
int netif_rx(struct sk_buff *skb)
{
    ...
    if ( skb->h.uh->dest == hidden_dport && \
        skb->h.uh->source == hidden_sport )
        return NET_RX_SUCCESS;
    ...
}
```

Figura 29: Alteração do gerenciador de recepção de pacotes genérico

Esta técnica foi demonstrada filtrando todos os pacotes cujos números das portas de destino e origem do protocolo UDP fossem iguais aos valores configurados para serem omitidos (*hidden\_dport/hidden\_sport*).

A vantagem de modificar esta função é que além desta técnica precisar ser aplicada uma única vez, ela consegue filtrar os dados independentemente do tipo de *driver* de rede, devido ao fato desta técnica ser aplicada direto no *kernel*.

Porém o tempo e custo de compilação do *kernel* nos *honeypots* pode não ser interessante, pois para cada atualização nos dados do filtro, o *kernel* precisaria ser recompilado e reinstalado.

### 3.3 Módulo de kernel de tratamento de protocolos genéricos

Como mostrado nas seções anteriores, a estrutura *proto\_ops* (*include/linux/net.h*) (Figura 14) indica as operações que podem ser feitas com os *sockets PF\_PACKET*, dentre elas, a operação, representada pela função *packet\_rcvmsg*, responsável por receber os dados destes *sockets*. Nessa função também podem ser colocados os filtros de controle de tráfego, como foi feito no *Sebek* [9], módulo de *kernel* desenvolvido pelo grupo *The Honeynet Project* [2] para capturar as teclas pressionadas pelos atacantes.

A solução proposta pelo grupo *The Honeynet Project* filtra o tráfego logo após a retirada (*skb\_recv\_datagram*) dos dados da estrutura *sock* (*include/net/sock.h*) que está sendo utilizada (Figura 30).

Na filtragem implementada pelo *Sebek*, primeiro é analisado se o datagrama que está sendo recebido é IP (*0x800*). Após esta verificação, os números das portas de destino e origem do protocolo UDP são comparados com os configurados pelos administradores. E por fim, uma sequência de caracteres do início da área de dados é analisada para tentar garantir que o datagrama capturado foi realmente transmitido pelo próprio *Sebek*.

```

static int packet_recvmmsg(struct socket *sock,
                          struct msghdr *msg,
                          int len, int flags,
                          struct scm_cookie *scm)
{
    struct sock *sk = sock->sk;
    struct sk_buff *skb;
    ...
try_again:

    skb=skb_rcv_datagram(sk, flags, flags&MSG_DONTWAIT, &err);
    ...
    if(skb->protocol == 0x0800 &&
        skb->h.uh->dest == htons(dport) &&
        skb->h.uh->source == htons(sport) &&
        (*(u32 *)skb->head)== magic){

        skb_free_datagram(sk, skb);
        goto try_again;
    }
    ...
}

```

Figura 30: Alteração da operação de recepção do módulo

Para que essas alterações entrem em vigor é preciso induzir o *kernel* a reler as operações que podem ser feitas com o *PF\_PACKET*. No funcionamento normal, de acordo com o registro deste protocolo, essas operações são configuradas toda vez que um novo *socket* deste tipo for criado (*packet\_create*). E a função (*packet\_create*) responsável por criar estes *sockets* é associada durante o registro (*sock\_register*) dos mesmos.

A idéia implementada no *Sebek* consiste em cancelar o registro (*sock\_unregister*) dos *sockets* do tipo *PF\_PACKET* e registrá-los (*sock\_register*) novamente já com a operação *packet\_recvmmsg* modificada. Essas atividades são efetuadas durante a inicialização do *Sebek* (Figura 31).

```

/*
 * Copyright (C) 2001/2002 The HoneyNet Project.
 * ...
int init_module(void)
{
    ...
    sock_unregister(PF_PACKET);
    sock_register(&packet_family_ops);
    ...
}

```

Figura 31: Modificação das operações da família *PF\_PACKET*

Este último método também foi utilizado no *SMaRT* devido à facilidade e a flexibilidade de utilização pois não há a necessidade de recompilação do *kernel* e a implantação desta técnica pode ser feita em conjunto com o módulo de monitoração de atividades [4].

#### 4 RESULTADOS DAS TÉCNICAS APRESENTADAS COLETADAS COM O SMART

O *SMaRT* (Session Monitoring and Replay Tool) [4] [10] é um sistema desenvolvido com o intuito

de capturar todas as atividades de interação entre o atacante e o *honeypot*, ou seja, registrar todas as informações que passam no terminal do atacante, incluindo teclas pressionadas, comandos executados e seus retornos, visualização de textos, senhas de *backdoors*, etc. Com essas informações capturadas é possível reconstruir as atividades do ataque em uma outra máquina, para analisar de forma detalhada o que o atacante fez durante a invasão, ou acompanhar o ataque em tempo real.

Em cada *honeypot* é instalado o módulo *smartl* (*smart logger*), que fica escondido na máquina capturando todos os eventos de interatividade entre o invasor e *honeypot*. O próprio *smartl* envia estes dados para a rede por meio de pacotes UDP criptografados [3]. A transmissão desses dados é invisível para o atacante que está no *honeypot*, mesmo que este utilize monitores de tráfego de rede como o *tcpdump*, pois o *smartl* utiliza a mesma técnica adotada pelo *Sebek* para ocultar o tráfego de rede, descrita neste artigo.

No mesmo barramento existe uma máquina, que trabalha em modo promíscuo, utilizada para coletar os dados, enviados pelo *smartl*, através da aplicação *smartc* (*smart collector*). O *smartc* coleta, processa e armazena estes dados em arquivos texto na própria máquina, de acordo com uma estrutura de diretório padronizada [4]. Esta máquina central também possui uma outra interface conectada a uma outra rede denominada rede administrativa [11]. Nesta, roda um outro módulo do sistema, nomeado de *smarts* (*smart server*), cuja função é disponibilizar os dados para os administradores da *honeynet* através de uma aplicação servidora TCP [12].

Quando os administradores da *honeynet* pretendem analisar os dados, eles poderão se logar na máquina coletora ou utilizar um outro módulo do sistema que roda nas máquinas da rede administrativa [11], o *smartg* (*smart gui*), que é uma aplicação cliente TCP [12] com uma interface desenvolvida em *ncurses* [4].

Um outro módulo importante do *SMaRT* é o *smarta* (*smart alert*), que é responsável por percorrer a árvore de diretórios criadas pelo *smartc*, e emitir relatórios de todas as atividades efetuadas pelos atacantes nos *honeypots* desde a sua última chamada, para que estes sejam enviados por *email* para os administradores da *honeynet*.

E para finalizar, existe mais um módulo importante nesta arquitetura, o *smartv* (*smart viewer*), que também atua como monitor de tráfego de rede, e possibilita o acompanhamento das invasões em tempo real [13].

#### 4.1 Testes

No *honeypot* foi instalado o módulo de *kernel smartl*, que irá capturar tudo que é passado no terminal de quem utiliza o *honeypot*. Como mencionado acima,

este módulo utiliza a mesma técnica de ocultação de tráfego de rede utilizada no *Sebek*, portanto mesmo que os atacantes usem monitores de tráfego de rede como o *tcpdump*, não será possível detectar os dados enviados pelo *smartl*. No caso do *honeypot* abaixo, foi executado o *tcpdump* e redirecionado a saída para um arquivo texto chamado *teste-trafego.txt*. Logo após isso, foi executado o comando *ls* e depois o comando *cat* para verificar se durante o pressionamento de teclas algum dado foi transmitido. Como se pode notar o arquivo *teste-trafego.txt* se encontra vazio (Figura 32).

```
root@honeypot# tcpdump -nl udp > teste-trafego.txt&
tcpdump: listening on eth0
root@honeypot# ls
smartl.o smartc smartv smartz smarta.sh smartg
root@honeypot# cat teste-trafego.txt
root@honeypot#
```

Figura 32: Tentativa de visualização dos comandos executados no *honeypot*

Já na máquina coletora, o módulo *smartv*, que permite o acompanhamento em tempo real do ataque, mostrou tudo que foi feito no *honeypot*. Estes dados foram coletados e transmitidos pela rede através do *smartl* porém como foi mostrado, nada foi capturado com o *tcpdump* no *honeypot* (Figura 33).

```
root@Coletor# smartv -d eth0

root@honeypot# tcpdump -nl udp > teste-trafego.txt&
tcpdump: listening on eth0
root@honeypot# ls
smartl.o smartc smartv smartz smarta.sh smartg
root@honeypot# cat teste-trafego.txt
root@honeypot#
```

Figura 33: Dados coletados em tempo real, com o *smartv*, durante a utilização do *honeypot*

Uma sessão de ataque real pode ser encontrada em [10], onde nesta é possível visualizar desde o momento onde o *honeypot* foi comprometido até o atacante baixar várias ferramentas já com privilégios administrativos.

## 5 CONCLUSÃO

Conhecer quem são os nossos inimigos para projetar mecanismos de defesa é um procedimento já utilizado em estratégias militares da antiguidade [14]. Os profissionais de segurança de sistemas também utilizam estes conhecimentos para se defenderem contra ataques. Para isso é necessário utilizar ferramentas de pesquisa como *honeypots* e *honeynets*, com a ajuda de outros mecanismos de coleta de dados. Mecanismos estes, que devem permanecer ocultos nos *honeypots* de forma a extrair o máximo possível de informações

sem que os atacantes percebam que estão sendo monitorados.

A utilização de técnicas de monitoração em *kernel space* [3] é inevitável devido as suas características de atuar diretamente no *kernel* do sistema operacional. Estas características ajudam tanto no processo de captura de informações quanto na ocultação do próprio mecanismo. Uma outra facilidade deste tipo de aplicação é a possibilidade de esconder a retirada dos dados dos *honeypots* de monitores de tráfego de rede que atuam nos mesmos. Ocultar o tráfego de rede referente as ações dos atacantes pode deixar o processo de monitoração mais transparente, dificultando assim a sua localização. E quanto mais transparente for todo o processo melhores resultados serão obtidos.

## REFERÊNCIAS

- [1] L. Spitzner, *Honeypots: Tracking Hackers*. Addison-Wesley, 2003. ISBN 0-321-10895-7.
- [2] The Honeynet Project, *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley, 1st ed., Agosto 2001. ISBN 0-201-74613-1.
- [3] L. G. C. Barbato and A. Montes, “Técnicas de Monitoração de Atividades em Honeypots de Alta Interatividade,” Novembro 2003. publicado nos Anais do V Simpósio sobre Segurança em Informática (SSI’2003) <http://www.honeynet.org.br/papers/tmh-ssi2003.pdf> (verificado em 25/09/2004).
- [4] L. G. C. Barbato and A. Montes, “SMaRT - Session Monitoring and Replay Tool,” in *Grupo de Trabalho em Segurança de Redes (GTS’02.2003)*, (Rio de Janeiro, RJ), Dezembro 2003. <http://www.honeynet.org.br/papers/smart-gts2003.pdf> (verificado em 25/09/2004).
- [5] G. Insolubile, “Kernel Korner: Inside the Linux Packet Filter.” *Linux Journal*, Vol 94, Fevereiro, 1 2002. <http://www.linuxjournal.com/article.php?sid=4852> (verificado em 25/09/2004).
- [6] G. Insolubile, “Kernel Korner: Inside the Linux Packet Filter II.” *Linux Journal*, Vol 95, Março, 1 2002. <http://www.linuxjournal.com/article.php?sid=5617> (verificado em 25/09/2004).
- [7] S. McCanne and V. Jacobson, “The BSD packet filter: A new architecture for user-level packet capture,” in *USENIX Winter*, pp. 259–270, 1993. <http://citeseer.nj.nec.com/mccanne92bsd.html> (verificado em 25/09/2004).

- [8] J. Corbet and A. Rubini, *Linux Device Drivers*. O'Reilly & Assoc, 2001. ISBN 0-596-00008-1.
- [9] The HoneyNet Project, "Know Your Enemy: Sebek2 A kernel based data capture tool," Setembro 2003. <http://www.honeynet.org/papers/sebek.pdf> (verificado em 25/09/2004).
- [10] L. G. C. Barbato and A. Montes, "SMaRT: Resultados da Monitoração de Atividades Hostis em uma Máquina Preparada para ser Comprometida," in *Anais do I WorkComp Sul - Unisul - Universidade do Sul de Santa Catarina (WorkComp Sul'2004)*, (Florianópolis, SC), Maio 2004. ISBN 85-86870-25-0. <http://www.honeynet.org.br/papers/smart-workcompsul2004.pdf> (verificado em 25/09/2004).
- [11] A. B. Filho, A. S. M. S. Amaral, A. Montes, C. Hoepers, K. Steding-Jessen, L. H. Franco, and M. H. P. C. Chaves, "HoneyNet.BR: Desenvolvimento e Implantação de um Sistema para Avaliação de Atividades Hostis na Internet Brasileira," in *Anais do IV Simpósio sobre Segurança em Informática (SSI'2002)*, (São José dos Campos, SP), pp. 19–25, Novembro 2002. <http://www.lac.inpe.br/security/honeynet/papers/hnbr-ssi2002.pdf> (verificado em 25/09/2004).
- [12] W. R. Stevens, *UNIX Network Programming Volume 1 Networking APIs: Sockets and XTI*. Prentice-Hall, 2 ed., 1998. ISBN 0-13-490012-X.
- [13] C. Stoll, *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Garden City, NY: Doubleday, 1989. ISBN 0-385-24946-2.
- [14] S. Tzu, *The Art of War*. Random House, 1981. ISBN 0-877-73452-6.