Hacking Linux Kernel Network Stack

Damian Put
(pucik@overflow.pl)

Wprowadzenie do “Kernel Hackingu”

- Definicja “Kernel Hackingu”
- Linux Kernel i jego modularna budowa
- Dostęp dzięki modułom
- Dostęp dzięki /dev/kmem
- LKM vs. /dev/kmem
- Tworzenie modułów
- Funkcja inicjalizująca i czyszcząca
- Przykładowy “Hello World”

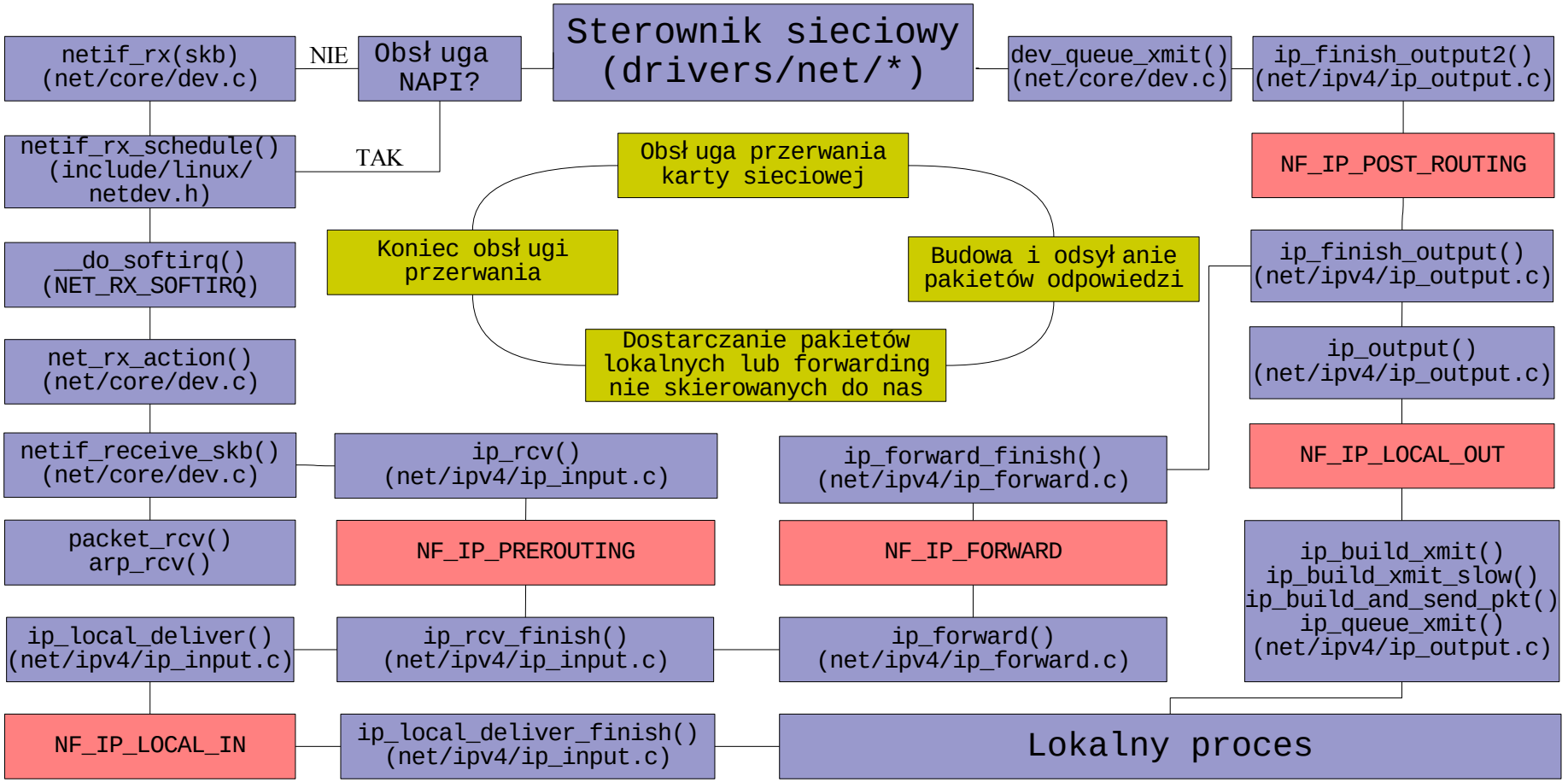
Hello World!

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

int  __init mod_init()
{
   printk("<1>Hello World!");
   return 0;
}
void __exit mod_exit()
{
}
module_init(mod_init);
module_exit(mod_exit);
// google -> “Kernel Hacking” :-)
```

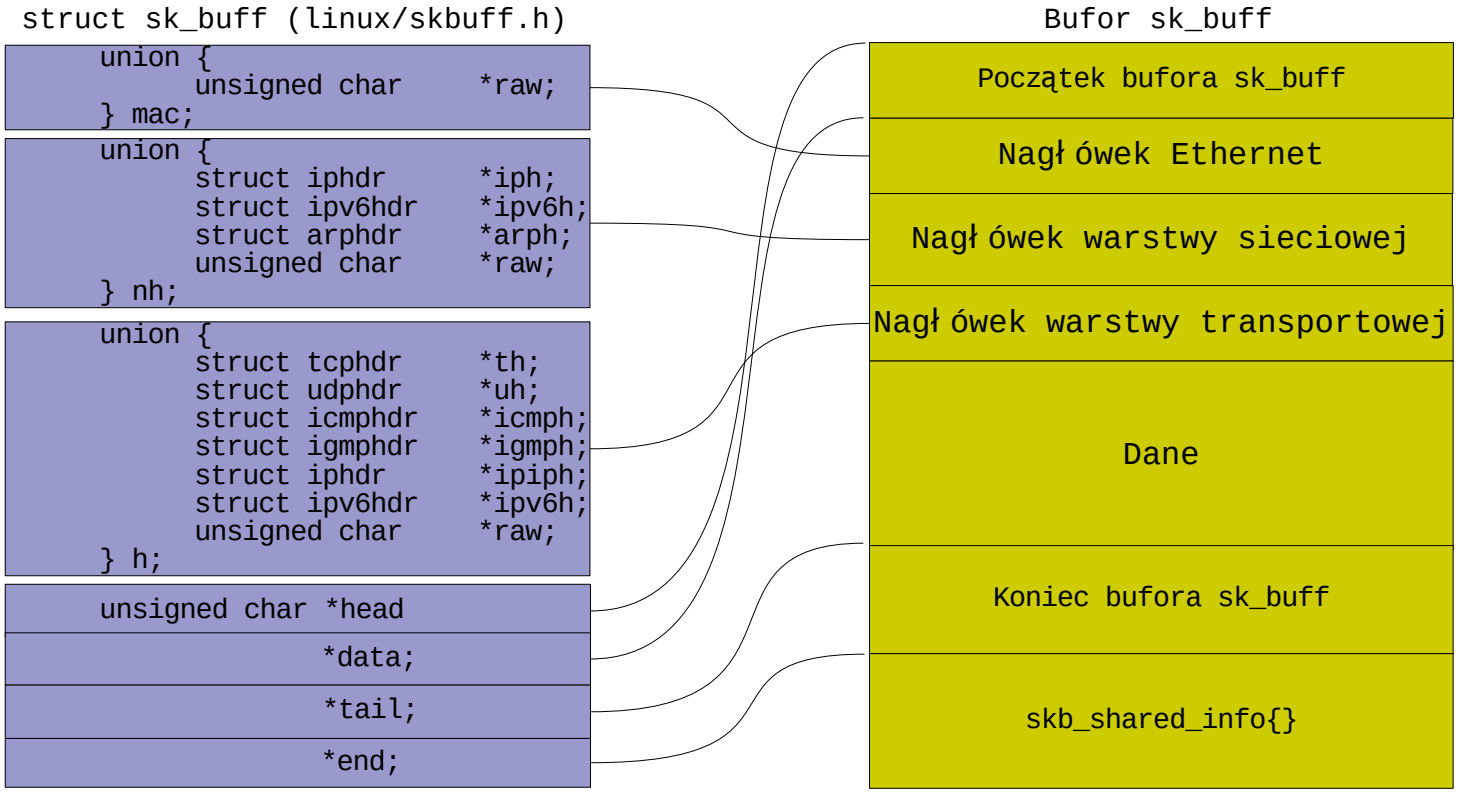
Możliwe zastosowania

- Sterowniki urządzeń
- Nowe funkcjonalności
- Wywołania systemowe (syscalls hooking)
- System plików (ukrywanie danych)
- Listy powiązane (ukrywanie procesów/modułów)
- Przerwania IRQ (śledzenie użytkownika)
- Stos sieci (“zabawa” z pakietami)
- itd...

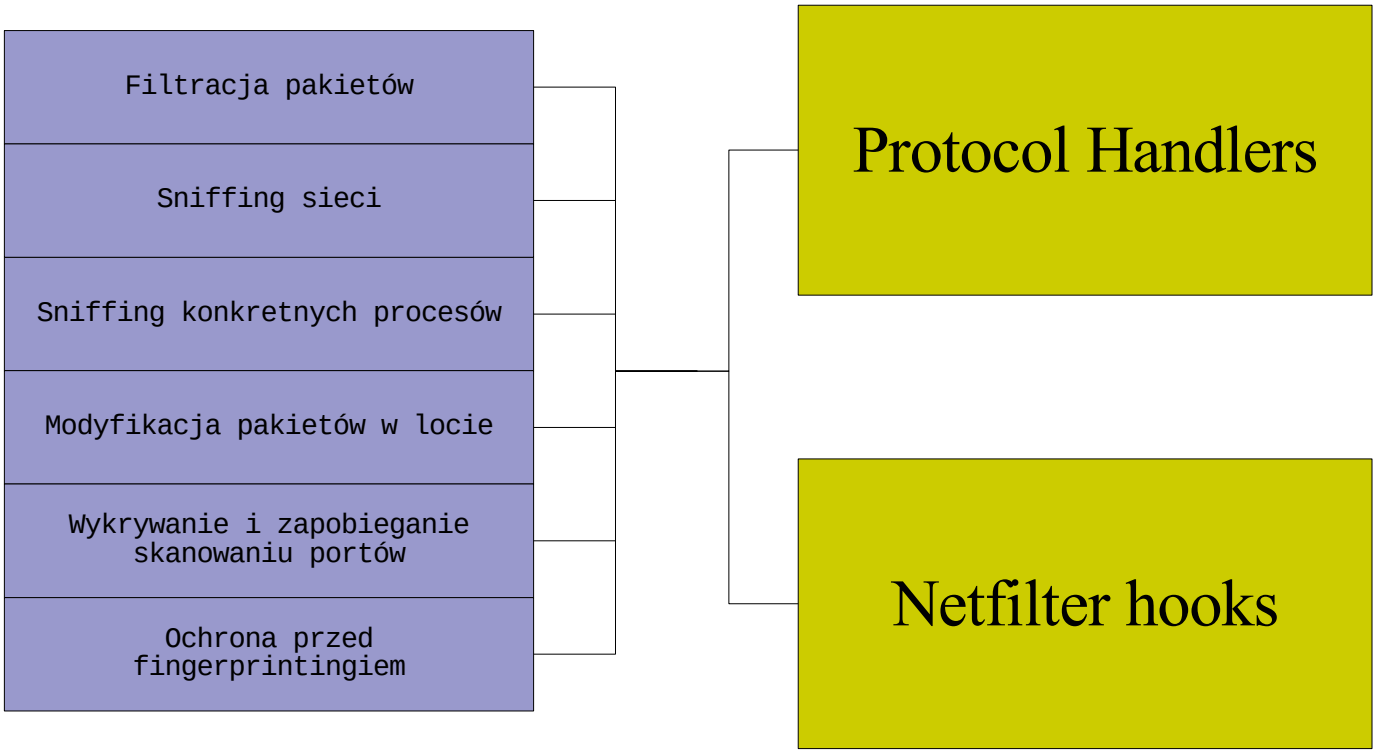
Obsługa pakietów przez jądro



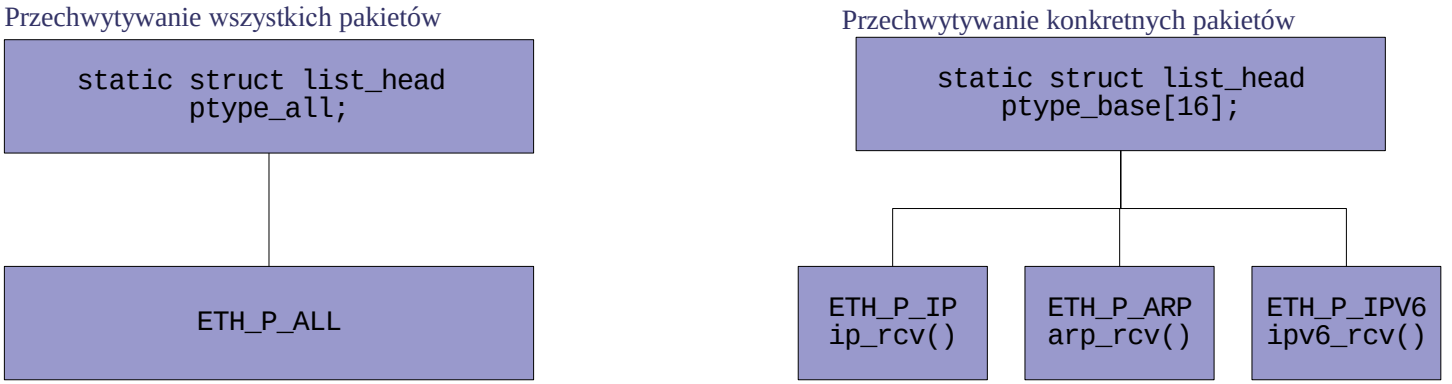
sk_buff - Socket Buffer Structure



Ingerencja w obsługę pakietów



Protocol handlers



- Możliwość tworzenia handlerów dla konkretnych protokołów
- Nowo dodane handlery, leżą przed wszystkimi innymi
- Prostota stosowania handlerów (jądra udostępnia odpowiednie funkcje)
- Wiele zastosowań (np. prosty sniffer, jedno z prymitywniejszych :)

Dodanie funkcji obsługi protokołu

```
/*
struct packet_type {
     unsigned short type; // Typ naszego handlera (ETH_P_ALL, ETH_P_IP etc.)
     struct net_device *dev; // Urządzenie ktore ma przechwytywać (NULL = wszystkie)
     int   (*func) (struct sk_buff *, struct net_device *, struct packet_type *);
     void  *af_packet_priv;
     struct list_head   list;
};
*/

struct packet_type hook_proto;

int __init module_init(void)
{
     hook_proto.type = htons(ETH_P_ALL);
     hook_proto.func = funkcja_obslugi;
     hook_proto.dev = NULL
     dev_add_pack (&hook_proto);
     return 0;
}

void __exit module_exit(void) {
     dev_remove_pack(&hook_proto);
}
```

Prosty sniffer pakietów #1

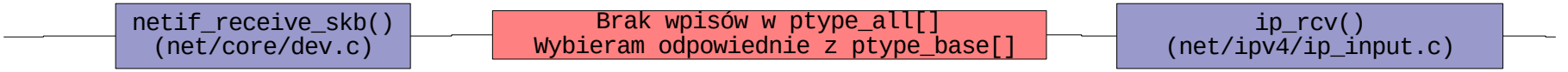
```
struct device *d;
struct packet_type sniffer_proto;

int __init cc_init(void)
{
    if (dev)
    {
        d = (struct device *)dev_get_by_name(dev);
        if (!d)
            printk("<1>Nie znalazlem urzadzenia %s!\n", dev);
    }
    memset(&sniffer_proto, 0, sizeof(struct packet_type));
    sniffer_proto.type = htons(ETH_P_ALL);
    sniffer_proto.func = (void*)sniffer_func;
    sniffer_proto.dev = (struct net_device *)d;
    dev_add_pack (&sniffer_proto);
    printk("<1>Sniffer zaladowany\n");
    return 0;
}
```

Prosty sniffer pakietów #2

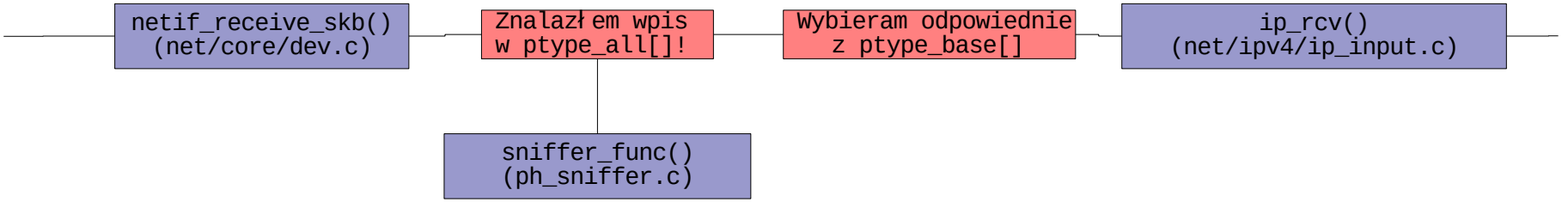
```
char *wzorce[] = {"Password", "PASS ", "password", "PASSWORD", "ScisleTajne", NULL};
int porty[] = {21,23,25,80,0};
int sniffer_func(struct sk_buff *skb, struct device *dv, struct packet_type *pt)
{
     char *data;
     struct tcphdr *tcph;
     int i, ok;
     if (!skb ) return 0;
     if (!(skb->nh.iph)) return 0;
     if (skb->nh.iph->protocol != IPPROTO_TCP)
          return 0; // Zależy nam tylko na pakietach TCP
     tcph = (struct tcphdr *)(skb->data + skb->nh.iph->ihl*4);
     /* Interesuja nas tylko pakiety skierowane na ktorys z tablicy porty[] */
     for(i=0; porty[i]; i++)
          if(porty[i] == htons(tcph->dest));
               ok=1;
     if(!ok)
          return 0; // Pakiety nie na nasze porty
     data = (char *)((int)tcph + (int)(tcph->doff * 4));
     /* Porównujemy dane z pakietu z naszymi stringami */
     for(i = 0; wzorce[i]; i++)
          if(strstr(data,  wzorce[i])){
               printk("<1>%s\n", data);
               return 0;
          }
     return 0;
}
```

Prosty sniffer pakietów #3

Obsługa pakietu przed dodaniem własnego handlera:



Obsługa pakietu po dodaniu własnego handlera:



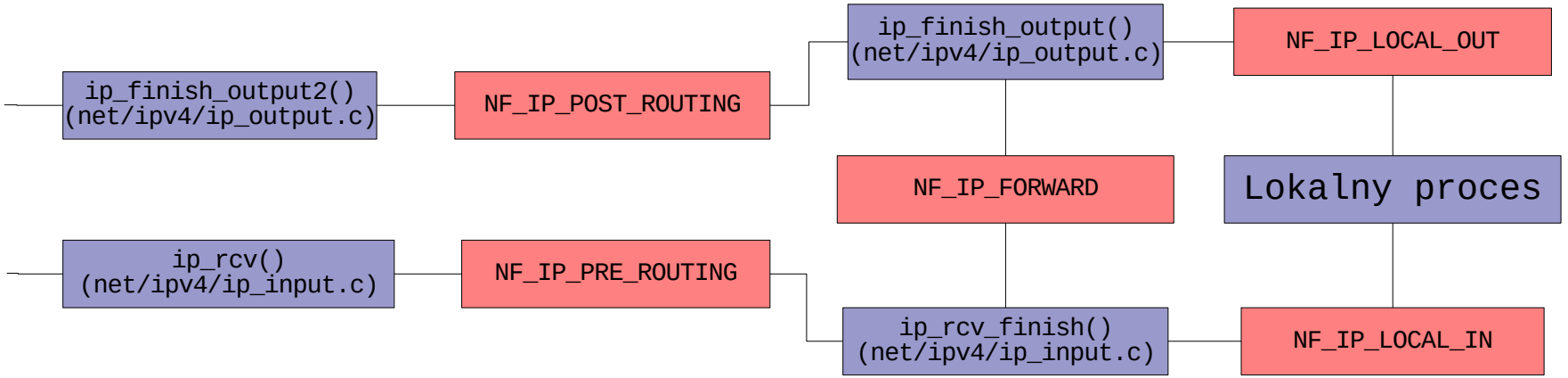
Protocol handlers - podsumowanie

- Dużo informacji w internecie
- Trudne (ale tylko z pozoru) do wykrycia (adresy ptype_all i ptype_base nie są eksportowane przez kernel)
- Przechwytywanie tylko przychodzących pakietów
- Istnieje o wiele bardziej elastyczna technika...
- ... która zwie się Netfilter hooks :-)

Netfilter hooks

- Czym jest netfilter?
- Netfilter jest stale rozwijany
- Większe możliwości niż protocol handler hooks...
- ... przy równie prostym sposobie użycia
- Podpinanie się w wielu miejscach
- Omijanie lokalnego firewalla
- Łatwość wykrycia (minus w przypadku rootkitów)

Punkty “zaczepu”



- NF_IP_PRE_ROUTING – Wszystkie przychodzące pakiety
- NF_IP_LOCAL_IN – Przychodzące pakiety przeznaczone dla nas
- NF_IP_FORWARD – Przychodzące pakiety nie przeznaczone dla nas
- NF_IP_LOCAL_OUT – Wychodzące pakiety od nas
- NF_POST_ROUTING – Wszystkie wychodzące pakiety

Dodanie własnego hooka

```
/*
struct nf_hook_ops
{
	struct list_head list;
	nf_hookfn *hook; // Funkcja obsługi hooka
	int pf; // Rodzina najczesciej PF_INET
	int hooknum; // Rodzaj hooka np. NF_IP_PRE_ROUTING
	int priority; // Priorytet jeśli więcej hooków (linux/netfilter_ipv4.h)
};
*/
unsigned int hook(unsigned int hooknum, struct sk_buff **sb, const struct net_device *in,
		const struct net_device *out, int (*okfn)(struct sk_buff *))
{
	return NF_DROP; // Odrzucaj wszystkie pakiety (NF_ACCEPT/NF_QUEUE/NF_STOLEN/NF_DROP)
}

struct nf_hook_ops nfho;

int __init module_init()
{
	nfho.hook = hook;  // Uchwyt do naszej funkcji
	nfho.hooknum  = NF_IP_PRE_ROUTING; // Wszystkie pakiety przychodzące
	nfho.pf       = PF_INET; // Rodzina
	nfho.priority = NF_IP_PRI_FIRST;  // Ustawiamy priorytet naszej funkcji na pierwszy
	nf_register_hook(&nfho); // Rejestracja funkcji
	return 0;
}

void __exit module_exit()
{
	nf_unregister_hook(&nfho); // Zwalnianie funkcji
}
```

Modyfikacja pakietów

- Możemy zmieniać zawartość każdego pakietu
- Wystarczy zmodyfikować odpowiednie struktury
- I wygenerować nową sumę kontrolną.
- Świetny sposób na normalizację pakietów wychodzących
- Ustawienie stałego TTL może zapobiedz rozpoznaniu topologi sieci
- A modyfikacja innych elementów skutecznie chroni przed fingerprintingiem.

Modyfikacja pakietów - przykład

```
unsigned int hook(unsigned int hooknum, struct sk_buff **sb, const struct net_device *in,
                  const struct net_device *out, int (*okfn)(struct sk_buff *))
{
      struct sk_buff *skb = *sb;
      struct iphdr *iph;
      struct tcphdr *tcph;
      int size, doff, csum;
      if (!skb ) return NF_ACCEPT;
      if (!(skb->nh.iph)) return NF_ACCEPT;
      if (skb->nh.iph->protocol != IPPROTO_TCP)
            return NF_ACCEPT;

      iph = skb->nh.iph;
      tcph = (struct tcphdr *)(skb->data + (iph->ihl * 4));

      tcph->window = htons(WINDOW);
      iph->ttl=TTL;

      /* Generujemy checksumy TCP/IP */
      size = ntohs(iph->tot_len) - (iph->ihl * 4);
      doff = tcph->doff << 2;
      skb->csum = 0;
      csum = csum_partial(skb->h.raw + doff, size - doff, 0);
      skb->csum = csum;
      tcph->check = 0;
      /* Checksuma TCP */
      tcph->check = csum_tcpudp_magic( iph->saddr, iph->daddr, size, iph->protocol,
            csum_partial(skb->h.raw, doff, skb->csum) );
      /* Checksuma IP */
      ip_send_check(iph);
      return NF_ACCEPT;
}
```

Idziemy krok dalej – zmiana tożsamości

- p0f czyli narzędzi do pasywnego fingerprintingu
- Rozpoznawanie systemów na podstawie ich “odcisków”
- Przyczyna - każdy stos sieci jest zaimplementowany nieco inaczej
- Pasywnego, czyli nie generującego żadnego ruchu
- Możemy oszukać p0fa w prosty sposób, modyfikując wszystkie pakiety wychodzące
- Będziemy musieli również ingerować w opcje TCP, a reszta po staremu :-)

Analiza sygnatury SYN p0fa #1

```
wwww:ttl:D:ss:OOO...:QQ:OS:Details
```

- wwww - Window size (rozmiar okna). Element nagłówka TCP
- ttl - Time To Live czyli czas życia pakietu.
- D - Don't fragment bit, określa czy pakiet można fragmentować czy nie
- ss - Całkowity rozmiar pakietu SYN
- OOO - Opcje nagłówka TCP (p0f.fp)
- QQ - Dziwne części pakietu, spowodowane błędną implementacją sieci
- OS - Rodzaj systemu (np. Linux, Solaris, Windows)
- Details - Rodzaj wersji systemu (np. 2.0.27 on x86)

Analiza sygnatury SYN p0fa #2

```
65535:64:1:60:M*,N,W2,N,N,T:Z:FreeBSD:5.1-current (3)
```

- Rozmiar okna pakietu wynosi 65535 (maksymalna wartość)
- Czas życia pakietu to 64 skoki.
- Pakiet może ulegać fragmentacji jeśli jest taka potrzeba
- Całkowita wielkość pakietu wynosi 60 bajtów
- Opcje TCP to kolejno: MSS o dowolnej wartości, NOP, Skala okna wynosi 2, NOP, NOP, Timestamp
- W pakiecie występują anomalie, pole ID nagłówka IP jest równe zero (Z)
- Rodzaj systemu to FreeBSD
- A dokładnie wersja 5.1-current

Podszywamy się pod FreeBSD

```
unsigned int hook(unsigned int hooknum, struct sk_buff **sb, const struct net_device *in,
                  const struct net_device *out, int (*okfn)(struct sk_buff *))
{
       struct sk_buff *skb = *sb;
       struct iphdr *iph; struct tcphdr *tcph;
       int size, doff, csum, tcplen,iplen, optlen, datalen, len;
       unsigned char *option; long *timestamp;
       unsigned int WINDOW = 65535;
       int TTL = 64, DF = 1, LEN = 60;
       unsigned char opcje[]=     "\x02\x04\x66\x66" // MSS o dowolnej wartoci (\0x66\0x66)
                                  "\x01" // NOP
                                  "\x03\x03\x02" // WSO o wartosci 2
                                  "\x01\x01" // Dwa NOPy
                                  "\x08\x10\x00\x00\x00\x00\x00\x00\x00\x00"; // Timestamp - musimy aktualizowac za kazdym razem

       ... // Ustawiamy odpowiednie wartości dla iph, tcph, tcplen, optlen etc.

       iph->ttl=TTL;
       iph->frag_off = DF ? htons(0x4000) : 0;

       timestamp = (long*)(opcje+12); *timestamp=htonl(jiffies);
       option=(char *)(tcph+1); optlen=LEN-40;
       memcpy(option, opcje, optlen);

       tcph->window=htons(WINDOW);
       iph->id =  0;

       tcph->doff=(sizeof(struct tcphdr)+optlen)/4;
       tcplen=tcph->doff <<2;
       iph->tot_len=htons(iplen+tcplen+datalen);
       skb->len=iplen+tcplen+datalen;

       ... // Generujemy checksumy

       return NF_ACCEPT;
}
```

p0fucker

- p0fucker – czyli narzędzie do oszukiwania p0fa
- Możliwość podszycia się pod dowolny system
- Łatwy w obsłudze
- To tylko PoC – korzysta jedynie z sygnatur SYN
- Pisane przezemnie – zdecydowanie NIE polecam :-)
- Używać na własną odpowiedzialność

p0fucker w akcji #1

```
    [root@overflow p0fucker]# ./p0fucker -o Sony
[1] Sony Playstation 2 (SOCOM?), win=32120, ttl=64, df=1, len=44 opt=M1460 ?[t/n] t
    [root@overflow p0fucker]# telnet localhost

    [root@overflow p0fucker]# ./p0fucker -o Symbian
    SymbianOS 7, win=33580, ttl=64, df=1, len=64 opt=NW1NNTNNSM1460 ?[t/n] n
    SymbianOS 6048 (Nokia 7650?), win=8192, ttl=255, df=0, len=44 opt=M1460 ?[t/n] n
[2] SymbianOS (Nokia 9210?), win=8192, ttl=255, df=0, len=44 opt=M536 ?[t/n] t
    [root@overflow p0fucker]# telnet localhost

    [root@overflow p0fucker]# ./p0fucker -l 2024 -o windows
    Windows 3.11 (Tucows), win=8192, ttl=32, df=1, len=44 opt=M1984 ?[t/n] n
[3] Windows 95, win=64240, ttl=64, df=1, len=64 opt=M1984NW0NNT0NNS ?[t/n] t
    [root@overflow p0fucker]# telnet localhost

    [root@overflow p0fucker]# ./p0fucker -l 1476 -o cisco
[4] Cisco 7200, Catalyst 3500, et, win=4128, ttl=255, df=0, len=44 opt=M1436 ?[t/n] t
    [root@overflow p0fucker]# telnet localhost

    [root@overflow p0fucker]# ./p0fucker -o plan9
[5] Plan9 edition 4, win=65535, ttl=255, df=0, len=48 opt=M1460W0N ?[t/n] t
    [root@overflow p0fucker]# telnet localhost

[6] [root@overflow p0fucker]# rmmod p0fucker_lkm
    [root@overflow p0fucker]# telnet localhost
```

p0fucker w akcji #2

```
   [root@overflow pucik]# p0f -i lo
   p0f - passive os fingerprinting utility, version 2.0.6
   (C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
   p0f: listening (SYN) on 'lo', 251 sigs (13 generic), rule: 'all'.

[1] 127.0.0.1:4333 - Sony Playstation 2 (SOCOM?)
        -> 127.0.0.1:23 (distance 0, link: ethernet/modem)

[2] 127.0.0.1:1968 - SymbianOS (Nokia 9210?)
        -> 127.0.0.1:23 (distance 0, link: sometimes modem)

[3] 127.0.0.1:3628 - Windows 95 (NAT!)
        -> 127.0.0.1:23 (distance 0, link: wireless/IrDA)

[4] 127.0.0.1:1918 - Cisco 7200, Catalyst 3500, et
        -> 127.0.0.1:23 (distance 0, link: IPSec/GRE)

[5] 127.0.0.1:4772 - Plan9 edition 4
        -> 127.0.0.1:23 (distance 0, link: ethernet/modem)

[6] 127.0.0.1:3646 - Linux 2.6.11 and newer (loopback) (up: 18 hrs)
        -> 127.0.0.1:23 (distance 0, link: sometimes loopback (2))
```

Ciekawe pomysły

- NmapFucker – czyli narzędzie do oszukiwania nmapa
- Ciche wyprowadzanie informacji (ukryte kanały)
- Implementacja trudnych do wykrycia backdoorów
- Restrykcja dla procesów przy przyjmowaniu połączenia
- Ukrywanie ruchu spod określonych adresów
- Rozbudowa p0fuckera o nowe funkcjonalności

Linki

- Opis protocol handlers - http://www.phrack.org/phrack/55/P55-12
- Opis netfilter hooks – http://www.phrack.org/phrack/61/p61-0x0d_Hacking_the_Linux_Kernel_Network_Stack.txt
- Dokumentacja netfiltra – http://www.netfilter.org/documentation
- Osbł uga pakietów na podstawie 2.6.10 – http://svn.gnumonks.org/trunk/doc/packet-journey-2.6.xml
- Osbł uga pakietów na podstawie 2.4 -http://svn.gnumonks.org/trunk/doc/packet-journey-2.4.sgml
- Opis struktury sk_buff i funkcji operującej na niej – http://svn.gnumonks.org/trunk/doc/skb-doc.sgml
- Ciekawe dokumenty na temat ukrytych kanał ów i detekcji rootkitów - http://invisiblethings.org/papers.html
- Kilka polskich artykuł ów na temat kernel hackingu – http://cc-team.org/index.php?name=artykuly&rid=3
- Kernel hacking za pomocą /dev/kmem - http://www.uebi.net/silvio/runtime-kernel-kmem-patching.txt
- Również /dev/kmem lecz tym razem bez użycia System.map - http://www.phrack.org/phrack/58/p58-0x07
- Odnajdywanie funkcji i struktur korzystając z /dev/kmem, polecam - http://www.ouah.org/p61_BONUS_BONUS.txt
- Publikacje grupy THC na temat kernel hackingu – http://www.thc.org/root/docs/loadable_kernel_modules/
- Strona narzędzia p0f – http://lcamtuf.coredump.cx/p0f.shtml
- Narzędzie p0fucker - http://overflow.pl/poc/p0fucker.tar.bz2

Pytania?

Dziękuję za uwagę