Содержание

Введение

Современные вычислительные системы сложно представить без поддержки интернет-сетей. На одном устройстве может быть доступ к различным подсетям по различным сетевым интерфесам и каналам передачи данных. Однако, со стороны пользовательского программного обеспечения может быть неудобно обращаться к различным сетевым интерфейсам. Агрегация нескольких сетевых интерфейсов в один может облегчить доступ к нескольким подсетям со стороны прикладных приложений.

Так как сетевой интерфейс может быть определен в системе только с помощью загружаемого модуля ядра, то данная работа будет основана на анализе и разработки такого загружаемого модуля.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать и реализовать загружаемый модуль ядра, реализующий создание виртуального интерфейса для распределения пакетов по уже существующим сетевым интерфейсам.

Для достижения цели курсовой работы необходимо решить следующие задачи:

- проанализировать пути решения задачи;
- спроектировать модуль, реализующий необходимую функциональность;
- реализовать спроектированный модуль;
- протестировать реализованный модуль.

Разрабатываемое ПО должно перенаправлять пакеты, пришедшие на виртуальный интерфейс по указанным интерфейсам в зависимости от IP назначения пакета.

1.2 Загружаемый модуль ядра Linux

Существует только один способов реализации требуемой функциональности — загружаемый модуль ядра.

Загружаемый модуль ядра — модули, позволяющие расширять и модифицировать ядро операционной системы его перекомпиляции. Из модуля ядра могут напрямую вызыватся системные функции. Модуль ядра может реализовывать драйвер устройства, файловую систему и выполняется на нулевом кольце защиты, имеющим максимальный уровень привилегий [1].

1.3 Сетевые интерфейсы и устройства

Для доступа к сетевым устройствам используются так называемые сетевые интерфейсы. Лги являются основной сетевой подсистемы Linux. Все сетевое взаимодействие в Linux происходит через сетевые интерфейсы. Любые данные, которые компьютер отправляет в сеть или получает из сети проходят через сетевой интерфейс.

Некоторые источники [2, 3] выделяют сетевые устройства в третий основной класс устройств в Linux, наравне с символьными и блочными.

Однако интерфейсы это не файлы устройств и их нет в каталоге /dev. Интерфейсы создаются динамически и не всегда связаны с сетевыми картами. Например интерфейс ppp0 - это интерфейс VPNa, организованного по протоколу PPTP, а интерфейс lo это виртуальная сетевая карта с адресом localhost (127.0.0.1). Такие интерфейсы называются виртуальными.

Таким образом сетевые интерфейсы скрывают детали реализации конкретного сетевого устройства, прикладное программное обеспечение, обращаясь к сетевому интерфейсу не учитывает детали реализации конкретных сетевых устройств. Однако в Linux существует общепринятая схема именования сетевых интерфейсов, состоящая из префикса типа сетевого устройства и заканчивающаяся номером иакого устройства. Примеры наименования интерфейсов:

- eth0 — первый сетевой интерфейс к карте Ethernet или картам WaveLan (Radio Ethernet);
- wlan0 — сетевой интерфейс wi-fi адаптера;
- lo — сетевой интерфейс к виртуальной сетевой карте с адресом localhost (127.0.0.1);
- eth1n3 — четвертый сетевой интерфейс второй группы к карте Ethernet или картам WaveLan (Radio Ethernet).

Интерфейсы создаются автоматически для каждого обнаруженного сетевого устройства при загрузке ядра ОС.

Каждый интерфейс характеризуется определёнными параметрами, необходимыми для обеспечения его нормального функционирования, и в частности для сетевого обмена данными с помощью стека TCP/IP. Некоторые параметры интерфейса:

1. IP-адрес;
2. маска подсети;
3. аппаратный адрес сетевого устройства, соответствующего интерфейсу.
4. eth1n3 — четвертый сетевой интерфейс второй группы к карте Ethernet или картам WaveLan (Radio Ethernet);

И сетевой интерфейс и драйвер сетевого устройства описываются большой структурой ядра ‘net_device‘, о которой сами разработчики, из-за смешения в ней разных уровней абстракции, отзываются как о "большой ошибке в коде ядра есть комментарий: "Actually, this whole structure is a big mistake".

1.4 net_device

Основной структурой, которую использует сетевая подсистема Linux является struct net_device (определена в <linux/netdevice.h>). Сама структура является слишком большой для полного приведения, поэтому рассмотрим только некоторые поля.

- char name[IFNAMSIZ] — имя устройста;
- unsigned long rmem_end , unsigned long rmem_start , unsigned long mem_end , unsigned long mem_start — информация о памяти устройства. Данные поля содержат начало и конец разделяемой памяти

устройства. Поля rmem служат для определения памяти для получения данных, а mem — для передачи. По соглашению поля end устанавливаются, поэтому end - start = общему количеству доступной памяти на устройстве;

- unsigned long base_addr — базовый адрес ввода-вывода сетевого интерфейса. Это поле, как и предыдущие, назначаются во время обнаружения устройства. Команда ifconfig может быть использована для отображения и модификации текущего значения;

- unsigned char irq — назначеный номер прерывания;

- unsigned char if_port — показывает, какой порт используется в устройствах с несколькими портами, например устройства с поддержкой как коаксиального (IF_PORT_10BASE2) Ethernet соединения, так и Ethernet соединения с помощью витой пары (IF_PORT_10BASET). Полный список известных типов портов определен в <linux/netdevice.h>;

- unsigned long state — состояние устройства. Это поле включает несколько флагов. Драйвер обычно не использует эти флаги напрямую, но с помощью специальных функций;

- void *priv — указатель, зарезервированный для пользовательских данных;

- struct net_device *next — указатель на следующее сетевое устройство в глобальном связанном списке сетевых устройств.

Большую часть информации, связанной с сетевыми интерфесами в структуре net_device заполняют существующие функции установки, определенные в <drivers/net/net_init.c>. Примеры таких функций:

1. void ether_setup(struct net_device *dev) — инициализирует поля для устройств Ethernet;

2. void ltalk_setup(struct net_device *dev) — инициализирует поля для устройств LocalTalk;

3. void fc_setup(struct net_device *dev) — инициализирует поля для волоконно-оптических устройств;

4. void fddi_setup(struct net_device *dev) — конфигурирует интерфейс для сети с Fiber Distributed Data Interface (распределенным интерфейсом передачи данных по волоконно-оптическим каналам, FDDI).

5. void hippi_setup(struct net_device *dev) — инициализирует поля для High-Performance Parallel Interface (высокопроизводительного параллельного интерфейса, HIPPI);

6. void tr_setup(struct net_device *dev) — выполняет настройку для сетевых интерфейсов token ring (маркерное кольцо).

Большинство устройств подходит под один из этих типов. Если требуется что-то уникальное, то необходимо определить следующие поля:

1. unsigned short hard_header_len — длина аппаратного заголовка;

2. unsigned mtu - MTU (Max transfer unit);

3. unsigned long tx_queue_len — максимальная длина очереди на отправку;

4. unsigned short type — аппаратный тип интерфейса;

5. unsigned char addr_len — длина аппаратного адреса;

6. unsigned char dev_addr[MAX_ADDR_LEN] — аппаратный адрес устройства (MAC).

7. unsigned short flags — флаги интерфейса;

8. int features — специальные аппаратные возможности.

Функции, с помощью которых система взаимодействует с устройством определены в структуре net_device_ops, определенной в <linux/netdevice.c>/ Часть структуры приведена ниже:

Листинг 1 – net_device_ops

```
struct net_device_ops {
        int (*ndo_init)(struct net_device *dev);
        void (*ndo_uninit)(struct net_device *dev);
        int (*ndo_open)(struct net_device *dev);
        int (*ndo_stop)(struct net_device *dev);
        netdev_tx_t (*ndo_start_xmit) (struct sk_buff *skb, struct
    net_device *dev);
        void (*ndo_change_rx_flags)(struct net_device *dev, int
    flags);
        void (*ndo_set_rx_mode)(struct net_device *dev);
        void (*ndo_set_multicast_list)(struct net_device *dev);
        int (*ndo_set_mac_address)(struct net_device *dev, void *
    addr);
        int (*ndo_validate_addr)(struct net_device *dev);
        int (*ndo_set_config)(struct net_device *dev, struct ifmap
    *map);
        int (*ndo_change_mtu)(struct net_device *dev, int new_mtu)
    ;
        void (*ndo_tx_timeout) (struct net_device *dev);
        struct net_device_stats* (*ndo_get_stats)(struct
    net_device *dev);
        /* Several lines omitted */
};
```

Из них минимально необходимы:

1. ndo_open — вызывается при открытии интерфейса;

2. ndo_close — вызывается при закрытии интерфейса;

3. ndo_start_xmit — вызывается при передачи пакета через интерфейс.

1.5 Виртуальные интерфейсы tun/tap

TUN и TAP — виртуальные сетевые драйверы ядра системы. Они представляют собой программные сетевые устройства, которые отличаются от обычных аппаратных сетевых карт.

TAP эмулирует Ethernet устройство и работает на канальном уровне модели OSI, оперируя кадрами Ethernet. TUN (сетевой туннель) работает на сетевом уровне модели OSI, оперируя IP пакетами. TAP используется для создания сетевого моста, тогда как TUN для маршрутизации.

Пакет, посылаемый операционной системой через TUN/TAP устройство обрабатывается программой, которая контролирует это устройство. Получение данных происходит через специальный файловый дескриптор, таким образом программа просто считывает данные с файлового дескриптора. Сама программа также может отправлять пакеты через TUN/TAP устройство выполняя запись в тот же файловый дескриптор. В таком случае TUN/TAP устройство доставляет (или «внедряет») такой пакет в сетевой стек операционной системы, эмулируя тем самым доставку пакета с внешнего устройства.

Не смотря на внешнюю схожесть TAP интерфейса с планируемым решением, нельзя просто строить решение на его основе из-за различной внутренней логики виртуальных интерфейсов.

1.6 Обработка пакетов на уровне сетевого интерфейса

В сетевых интерфейсах существует очередь обрабатываемых пакетов. На программном уровне следующий пакет из буфера пакетов, требующих обработки представлен структурой sk_buff.

На рисунке 1 проиллюстрирован процесс получения интернет-пакетов. Сетевые интерфейсы работают с пакетами 2-го и 3-го уровня модели OSI.

Для применения к поставленной задачи необходимо как отправлять на несколько интерфейсов, так и считывать пакеты, отправленные в ответ. Для этого, как видно из схемы, необходимо вклинится в процесс обработки пакета. Linux позволяет добавить обработчик входящих пакетов с помощью функции netdev_rx_handler_register.

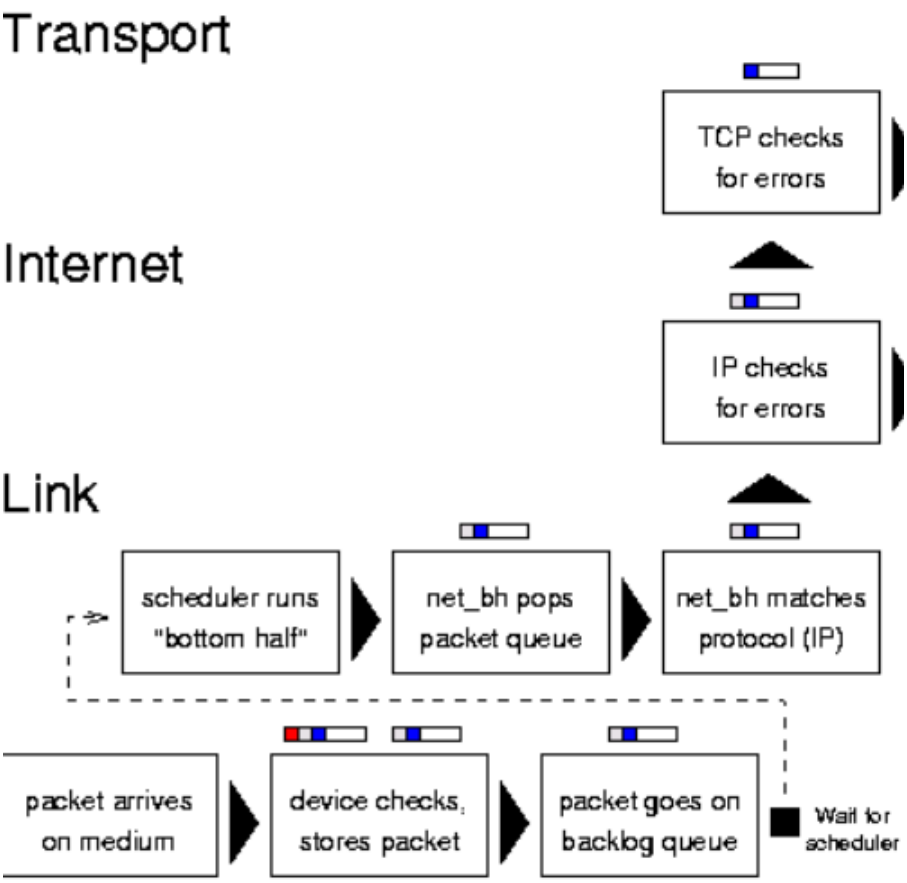

Рисунок 1 – Обработка входящего пакета

Исходящий же пакет можно перенаправить на обработку в другой интерфейс с помощью подмены поля dev в структуре sk_buff.

2 Конструкторский раздел

2.1 Требования к программному обеспечению

Программное обеспечение состоит из виртуального сетевого, реализованного в виде загружаемого модуля ядра, который распределяет приходящие на него пакеты по существующим сетевым устройствам.

2.2 Проектирование загружаемого модуля

Общая структура загружаемого модуля представлена схемой на рисунке 2.

Рисунок 2 – Общая структура загружаемого модуля

Схема алгоритма обработки принимаемого пакета на одном из интерфейсов, скрываемых виртуальным представлена на рисунке 3.

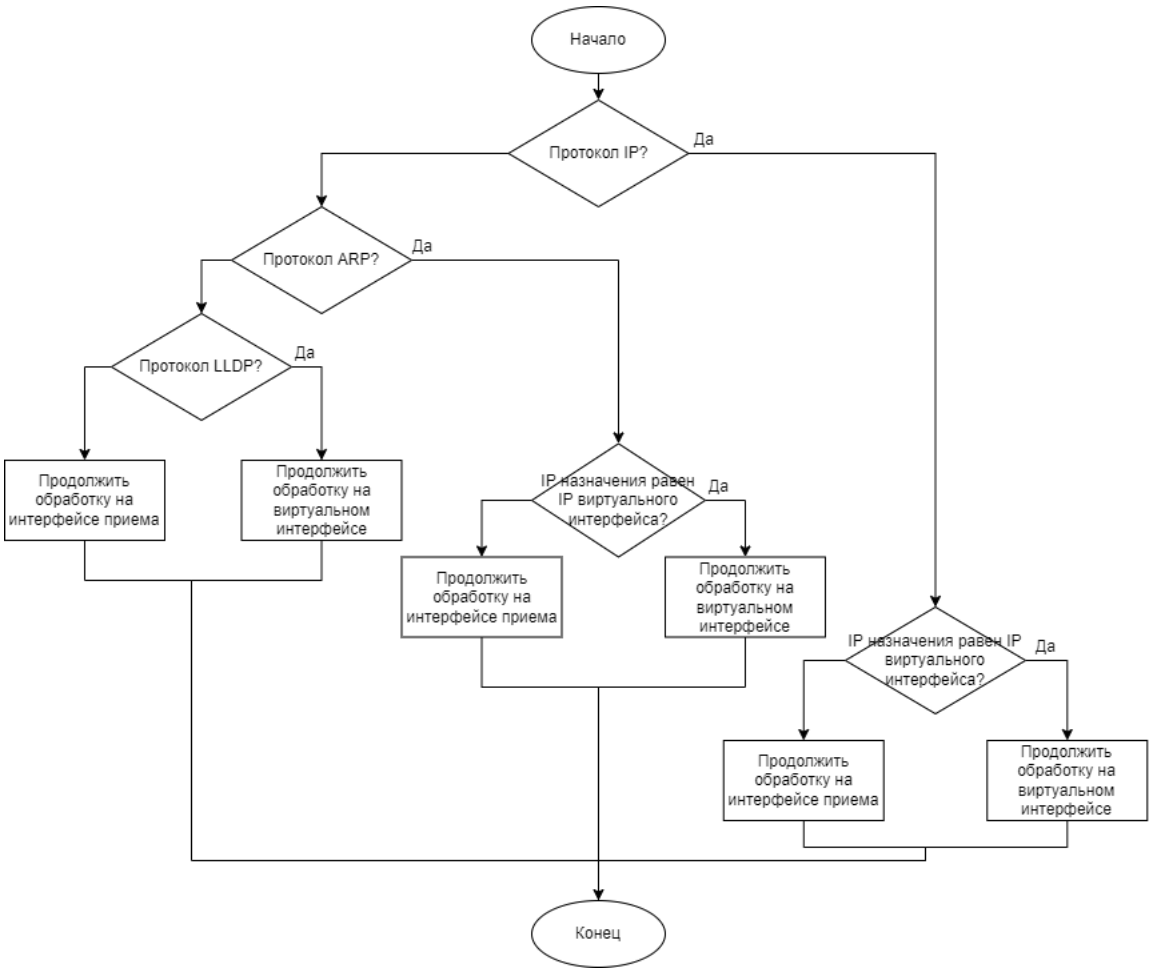

Рисунок 3 – Схема алгоритма обработки принимаемого пакета

Схема алгоритма обработки отправляемого пакета, поступившего на виртуальный интерфейс представлена на рисунке 4.

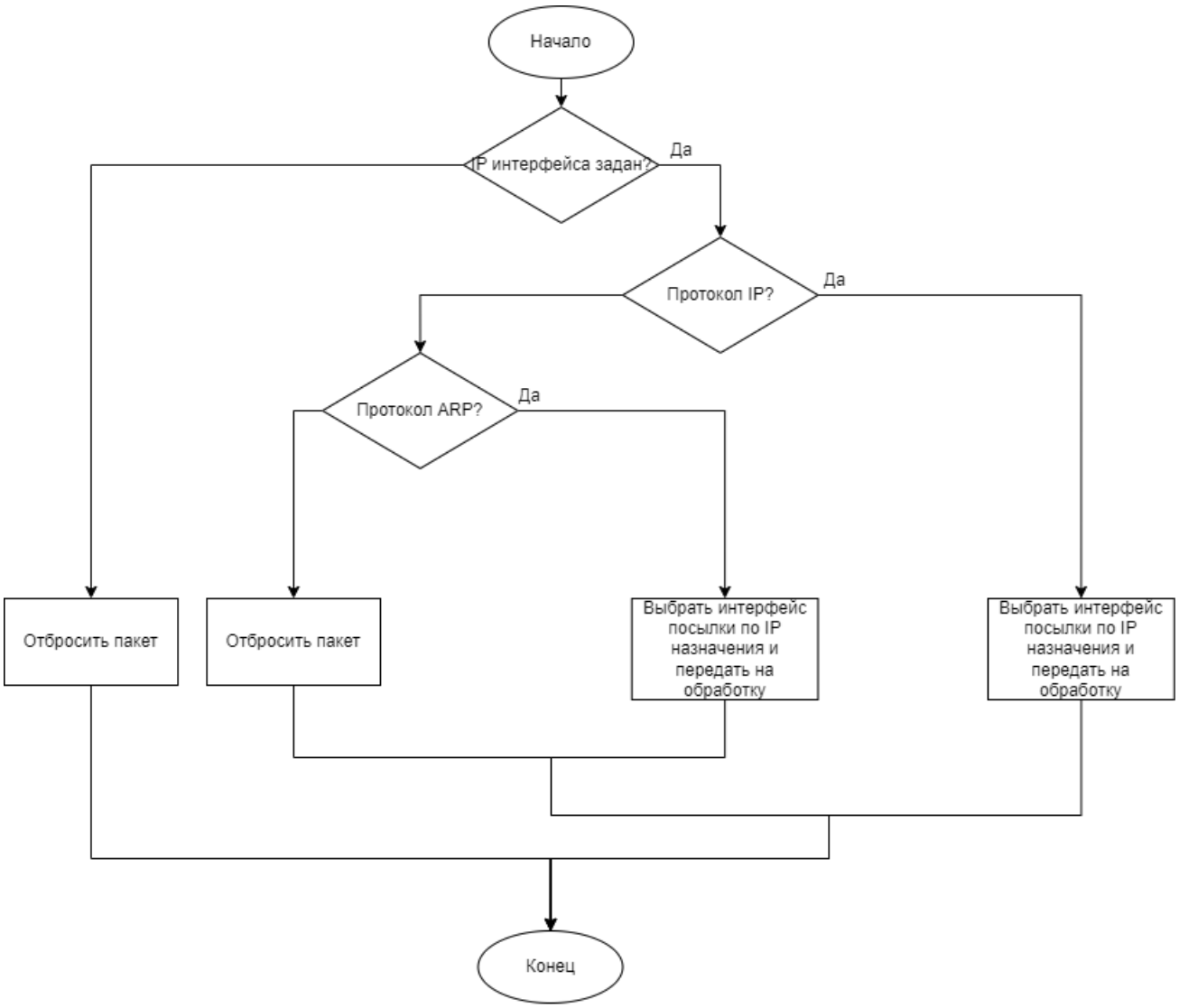

Рисунок 4 – Схема алгоритма обработки отправляемого пакета

3 Технологический раздел

3.1 Выбор языка программирования и среды программирования

В качестве языка программирования для реализации поставленной задачи был выбран язык C. Он является языком реализации модулей ядра и самого ядра ОС Linux. В качестве компилятора был использован компилятор gcc. Средой разработки был выбран текстовый редактор Visual Studio Code.

3.2 Описание реализации

Для реализации хранения нескольких интерфейсов для распределения пакетов было принято решение хранить в связанном списке ссылки на структуры net_device и дополнительную информацию об этих интерфейсах. Голова связанного списка хранится в приватной зоне памяти создаваемого виртуального интерфейса. На листинге 2 Приведена структура узла вышеупомянутого связанного списка.

Листинг 2 – lst example

```
struct interfaces {
    u32 address; // адрес подсети
    u32 mask;    // маска подсети
    struct net_device *device; // ссылка на интерфейс
    struct interfaces *next;   // следующий узел
};
```

Для поиска подходящего интерфейса по связанному списку была написана функция find_device_sub. В ней осуществляется проверка принадлежности адреса назначения к подсети по этому интерфейсу. Она представлена на листинге 3.

Листинг 3 – lst example

```
static struct net_device *find_device_sub(struct interfaces *subs,
    u32 addr)
{
    struct net_device *device = NULL;
    while (subs && !device)
    {
        u32 res = apply_mask(subs->mask, addr);
        if (res == subs->address)
        {
            device = subs->device;
        }
        else
        {
            subs = subs->next;
        }
    }

    return device;
}
```

В приватной области памяти создаваемого виртуального сетевого интерфейса создается структура priv, хранящая статистику интерфейса и начало связанного списка интерфейсов для распределения пакетов. На листинге 4 прелставлена структура priv.

Листинг 4 – lst example

```
struct priv
{
    struct net_device_stats stats;
    struct interfaces *next;
};
```

Функция обработки входящего пакета в создаваемый сетевой интерфейс представлена на листинге 5. Видно, что при нахождении подходящего интерфейса меняется поле dev структуры sk_buff, а затем буфер отправ-

ляется на дальнейшую обработку функцией dev_queue_xmit.

Листинг 5 – lst example

```
static netdev_tx_t start_xmit(struct sk_buff *skb, struct
    net_device *dev)
{
     struct in_device *in_dev = child->ip_ptr;
     struct in_ifaddr *ifa = in_dev->ifa_list;
     if (ifa)
     {
          struct priv *priv = netdev_priv(dev);
          priv->stats.tx_packets++;
          priv->stats.tx_bytes += skb->len;
          LOG("GET IP %d, %s", get_ip(skb), strIP(get_ip(skb)));
          struct net_device *device = find_device_sub(priv->next,
    get_ip(skb));
          if (device)
          {
               skb->dev = device;
               skb->priority = 1;
               dev_queue_xmit(skb);
               LOG("OUTPUT: injecting frame from %s to %s. Tarhet IP:
     %s", dev->name, skb->dev->name, strIP(get_ip(skb)));
               return NETDEV_TX_OK;
          }
     }
     return NETDEV_TX_OK;
}
```

На листинге 6 Показано заполнение структуры net_device_ops и определение оставшихся необходимых функций.

Листинг 6 – lst example

```
static int open( struct net_device *dev ) {
    netif_start_queue( dev );
    LOG( "%s: device opened", dev->name );
```

```
    return 0;
}

static int stop(struct net_device *dev)
{
    netif_stop_queue(dev);
    LOG("%s: device closed", dev->name);
    return 0;
}

static struct net_device_stats *get_stats(struct net_device *dev)
{
    return &((struct priv *)netdev_priv(dev))->stats;
}

static struct net_device_ops crypto_net_device_ops = {
    .ndo_open = open,
    .ndo_stop = stop,
    .ndo_get_stats = get_stats,
    .ndo_start_xmit = start_xmit,
};
```

Для отсутствия блокировки используемых интерфейсов необходимо передавать в созданный интерфейс только пакеты, предназначенные ему. Для этого в функции перехвата производится проверка на соответствие IP. На листинге 7 представлена функция перехвата.

Листинг 7 – lst example

```
tatic rx_handler_result_t handle_frame(struct sk_buff **pskb)
{
    struct sk_buff *skb = *pskb;
    struct in_device *in_dev = child->ip_ptr;
    struct in_ifaddr *ifa = in_dev->ifa_list;
    if (!ifa)
    {
```

```
8           return RX_HANDLER_PASS
9       }
10      u32 child_ip = ifa->ifa_address;
11      if (skb->protocol == htons(ETH_P_IP))
12      {
13          struct iphdr *ip = ip_hdr(skb);
14          LOG("INCOME: IP to IP=%s", strIP(ip->daddr));
15          if (!ifa || ip->daddr != child_ip)
16          {
17              return RX_HANDLER_PASS;
18          }
19      }
20      else if (skb->protocol == htons(ETH_P_ARP))
21      {
22          struct arphdr *arp = arp_hdr(skb);
23          struct arp_eth_body *body = (void *)arp + sizeof(struct
    arphdr);
24          int i, ip = child_ip;
25          LOG("INCOME: ARP for IP=%s", strAR_IP(body->ar_tip));
26          for (i = 0; i < sizeof(body->ar_tip); i++)
27          {
28              if ((ip & 0xFF) != body->ar_tip[i])
29                  break;
30              ip = ip >> 8;
31          }
32          if (i < sizeof(body->ar_tip))
33              return RX_HANDLER_PASS;
34      }
35      else if (skb->protocol != htons(0xCC88))
36      {
37          return RX_HANDLER_PASS;
38      }
39
40      LOG("INCOME: PASS");
41      struct priv *priv = netdev_priv(child);
```

```
42      priv->stats.rx_packets++;
43      priv->stats.rx_bytes += skb->len;
44      skb->dev = child;
45      return RX_HANDLER_ANOTHER;
46  }
```

3.3 Makefile

В листинге 8 приведен файл Makefile, используемый для сборки модуля ядра.

Листинг 8 – Makefile

3.4 Результат работы

Список литературы

1. ANDREW S. TANENBAUM HERBERT BOS / Modern Operating Systems FOURTH EDITION [Электронный ресурс]479-480 (дата

2. TEST [Электронный ресурс] URL: https://www.cisco.com/c/en/us/products/colla nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html (дата обращения: 09.06.2021).

Рисунок 5 – Example images

$$timeout = RTT_{average} + 4dev_{average} \tag{1}$$

Листинг 9 – lst example

Таблица 1 – Пример таблицы.

	UDP	TCP	MPTCP
Гарантия надежности доставки	-	+	+
Гарантия упорядоченности данных	-	+	+
Относительная скорость работы	1 место, за счет минимизации протокола	2 место, существенно проседает	3 место, ещё более тяжеловесен чем TCP
Возможность работы с несколькими каналами	-	-	+