«Научно-технический центр Метротек»

Лабораторная работа №2

Тема: Реализация NetFlow-сенсора

Стажёр: Бронников Егор Игоревич

<bronnikov.40@mail.ru>
<t.me/endygamedev>

Санкт-Петербург

2022

Содержание

Постановка задачи

Цель задачи: создать программу NetFlow-сенсор, поддерживающую NetFlow export protocol версии 9.

Требования:

1. ПО должно работать на ПК под управлением Debian GNU\Linux (версии 10 и новее).
2. Для реализации использовать язык программирования C.
3. Сборка должна осуществляться GNU Toolchain.
4. Дистрибуция должна осуществляться при помощи deb-пакета.
5. Сенсор должен иметь следующие параметры запуска:
    1. имя сетевого интерфейса, на котором вести учёт трафика
    2. IP-адрес и номер UDP-порта хоста с NetFlow-коллектором, куда отправлять пакеты с данными по статистике, формат IP:порт (например, 192.168.0.2:9995)
6. Основной приоритет при разработке: постараться максимизировать нагрузку, которую может обработать программа, не пропуская пакеты.

Описание протокола NetFlowV9: https://www.ietf.org/rfc/rfc3954.txt

Определение потока: поток (flow) это пакеты, имеющие одинаковые поля:

- IP адрес источника;
- IP адрес получателя;
- Для TCP/UDP пакетов:
    - TCP/UDP порт источника;
    - TCP/UDP порт получателя;
- Для ICMP пакетов:
    - ICMP код;

  - ICMP тип;
- Протокол L4 (поле IP Protocol Number);
- IP ToS.

Сенсор должен запускаться с указанием сетевого интерфейса, на котором вести учёт трафика, без подтверждения пользователя приступать к учёту и вести его до прерывания работы пользователем.

Пакет NetFlow должен содержать следующий набор полей:
- IN_BYTES,
- IN_PKTS,
- FLOWS,
- PROTOCOL,
- SRC_TOS,
- TCP_FLAGS,
- L4_SRC_PORT,
- IPV4_SRC_ADDR,
- INPUT_SNMP,
- L4_DST_PORT,
- IPV4_DST_ADDR,
- LAST_SWITCHED,
- FIRST_SWITCHED,
- ICMP_TYPE,
- FLOW_ACTIVE_TIMEOUT,
- FLOW_INACTIVE_TIMEOUT,
- IPV4_IDENT,
- IN_SRC_MAC,
- IN_DST_MAC,
- IF_NAME.

Проверка работы

Сенсор отправляет данные в коллектор. В качестве коллектора для тестирования можно использовать, например, `nfcapd` из пакета `nfdump`. Отправять отладочный трафик на интерфейс можно при помощи утилиты `tcpreplay` из одноимённого пакета. Для генерации дампа отладочного трафика можно использовать: https://github.com/cslev/pcap_generator.

Вместе с кодом нужно предоставить информацию о максимальной нагрузке, которую может выдержать предложенная реализация.

Рекомендации

Пример проекта, который можно из исходников собрать в deb-пакет: https://gitlab.com/vgeo89/examples/-/tree/main/deb_package

Информация

- http://xgu.ru/wiki/NetFlow
- https://www.ietf.org/rfc/rfc3954.txt
- https://www.cisco.com/en/US/technologies/tk648/tk362/technolo
white_paper09186a00800a3db9.html
- https://habr.com/ru/company/metrotek/blog/327894/

Приложение А

Coding Style

C

В сторонних проектах с собственным описанным стилем оформления кода следует придерживаться правил этого стиля:

- U-Boot: https://www.denx.de/wiki/U-Boot/CodingStyle
- Linux: https://www.kernel.org/doc/html/v4.10/process/coding-style.html

В сторонних проектах без описанного стиля следует оформлять наш код по аналогии с остальными исходниками проекта.

В наших собственных проектах за основу взят стиль, принятый в linux с некоторыми изменениями.

Отступы

В качестве отступа используется один таб шириной в четыре пробела.

```
int main(void)
{
    int a = 0;
    return a;
}
```

Лишние табы и пробелы в конце строк следует удалять.

Длина строк

Следует избегать превышения ограничения в 80 символов на строку. Если выражение не помещается в 80 символов, его следует разделить

на части. При этом желательно, чтобы каждый аргумент функции находился в отдельной строке. Пример:

```
return_code = my_structure->callback_with_arguments(argument_number_1,
                                                    argument_number_2);
```

Строки, выводимые программой на экран, не следует сокращать при превышении ими ограничения.

Скобки

В условиях, циклах и объявлениях структур открывающая фигурная скобка не переносится на следующую строку. Закрывающая переносится:

```
while (a > b) {
    a--;
    b++;
    if (a == b) {
        do_something();
    }
}
```

Закрывающая фигурная скобка располагается на том же уровне отступов, что и начало всей конструкции.

В объявлениях функций открывающая фигурная скобка переносится на следующую строку:

```
int main(int argc, char **argv)
{
    return 0;
}
```

В условиях и циклах перед открывающей круглой скобкой ставится пробел. После закрывающей скобки ставится пробел:

```
if (a == 0) {
    return a;
}
```

Фигурные скобки должны присутствовать даже если в блоке всего одно выражение.

Пробелы

Бинарные и тернарные операторы окружаются пробелами:

```
= + - < > * / % | & ^ <= >= == != ? :
```

После унарных операторов не ставится пробел:

```
& * + - ~ ! sizeof typeof alignof __attribute__ defined
```

Пробел не ставится после префиксных инкремента и декремента:

```
++i
--i
```

и перед постфиксными:

```
i++
i--
```

Пробелы не ставятся вокруг операторов структур:

```
my_struct.element
my_struct->element
```

При объявлении указателя оператор * идет перед именем указателя, а не после его типа:

```
void *buffer
```

Имена и объявления

В именовании функций, переменных и типов используется *snake_ case*. Примеры:

```
void my_function(int my_argument);
struct my_struct *s;
```

В именах желательно стараться избегать сокращений, из-за которых становится не ясно назначение сущности. Например, вместо **dctl()** использовать более полное имя **device_control()**.

Объявлять переменные в функции желательно рядом с выражениями, их использующими, если функция достаточно длинная (от 10 строк), либо в начале функции, если короткая (до 10 строк). Например, переменную-счётчик для цикла for следует объявлять так:

```
for (int i = 0; i < 3; ++i) {
    actions();
}
```

Также нужно следить за количеством переменных в одной функции. Если их число превышает пять, то следует задуматься о декомпозиции функции.

Функции

Функции должны быть как можно короче и выполнять как можно меньше действий. Длинные функции следует разбивать на подфункции, действия внутри которых связаны по смыслу. Пример:

```
int collect_data_and_calculate_result(struct program_context *c)
{
    void *data = collect_data(c);

    return calculate_result(data);
}

int init_and_run_program(struct arguments *a){
    struct program_context *c = init_context(a);

    return collect_data_and_calculate_result(c);
```

```
}
```

Структуры перечисления

Их объявления должны выглядеть следующим образом.
Структуры только с именем:

```
struct my_struct_name {
    int i;
    int j;
    void *buffer;
};
```

Структуры с typedef должны иметь суффикс _t в имени типа:

```
typedef struct my_struct {
    int i;
    int j;
    void *buffer;
} my_struct_t;
```

Перечисление объявляются аналогичным образом. Все варианты перечисления должны именоваться в верхнем регистре:

```
typedef enum {
    ADDRESS_7BIT,
    ADDRESS_10BIT,
} addressing_mode_t;
```

Макросы

Желательно избегать написания макросов. Особенно вложенных, так как это приводит к сложностям в отладке.

Макросы именуются с использованием только верхнего регистра. Макрос и отдельно его входные параметры должны быть окружены круглыми скобками:

```
#define MIN(x, y)     ((x) < (y) ? (x) : (y))
```

Макросы с несколькими выражениями должны быть заключены в блок `do {} while(0)`:

```
#define DO_ACTION(a, b)                         \
                do {                            \
                    if (do_first(a) >= 0) {     \
                    do_next(b);                 \
                    }                           \
                } while(0)                      \
```

Использование goto

`goto` можно использовать для обработки ошибок. Переходя должны осуществляться только в пределах одной функции. Пример:

```
int configure_device(void)
{
    int ret = 0;
    struct my_device *d = malloc(sizeof(my_device));
    if (!d) {
        return -ENOMEM;
    }

    ret = init_stage_first(d);
    if (ret != 0) {
        goto err_free;
    }

    ret = init_stage_last(d);
    if (ret != 0) {
        goto err_free;
    }

    return 0;

err_deinit:
```

```
    deinit_stage_first(d);
err_free:
    free(d);
    return ret;
}
```

Комментарии

Комментарии следует использовать только в качестве документации и для пояснения каких-то не очевидных специфичных случаев.

Функции

Документировать следует функции, поведение, входные и выходные параметры которые не очевидны. Пример:

```
/*
 * Configures an I2C bus.
 *
 * addr_lenght should be ADDRESS_7BIT or ADDRESS_10BIT.
 * speed should be SPEED_100MHZ or SPEED_400MHZ.*
 * Returns 0 on success, -1 otherwise.
 */
int configure_i2c_bus(int bus_number, int addr_length, int speed)
```

Не следует документировать очевидные функции. Например, в достаточно очевидны назначение и принцип работы функции. Пример:

```
int sum_int(int a, int b)
{
    return a + b;
}
```

Магические числа

Если назначение числового значения не очевидно из имени переменной или макроса, которым оно присваивается, то это значение следу-

ет снабдить комментарием. Пример:

```
#define DEVICE_CONTROL_REGISTER_ADDR 0x4
/* Set required device mode on startup. See device datasheet page
   100, table 20. */
#define DONTROL_REGISTER_DATA 0x4f7a
```

Заголовочные файлы

Должны обязательно содержать защиту от повторного включения:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
All the contents of the header file here
#endif
```

либо

```
#pragma once
Content
```

Содержимое может включать:

- дополнительные `include`
- объявление макросов
- объявление констант
- прототипы функций

но реализации функций должны содержаться только в файлах `.c`

Python

См. https://peps.python.org/pep-0008/