Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Модуль ядра для подмены пользовательских данных, передаваемых на съёмные USB-носители»

Студент ИУ7-75Б (Группа) _____ (Подпись, дата) Смирнов П. И. (И. О. Фамилия)

Руководитель курсовой работы _____ (Подпись, дата) Рязанова Н. Ю. (И. О. Фамилия)

2026 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Программно-аппаратные комплексы предотвращения утечек данных (DLP — Data Leak Prevention) включают в себя механизмы для нейтрализации одной из самых распространенных угроз информационной безопасности — несанкционированного копирования информации на портативные USB-устройства [1].

Основу для решения этой задачи в ядре Linux составляют драйверы устройств, а также механизмы перехвата управления у функций ядра. Данная работа посвящена созданию и исследованию загружаемого модуля ядра для подмены данных, передаваемых на USB-накопители, с использованием перехвата управления у функций.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать модуль ядра для подмены пользовательских данных, передаваемых на USB-накопители. Для решения поставленной задачи необходимо:

1. провести анализ функций и структур ядра для работы с USB-устройствами;
2. провести анализ методов перехвата управления у функций ядра;
3. провести анализ методов подмены и повреждения информации;
4. провести анализ работы файловой системы FAT32;
5. провести анализ способа синхронизации кэша процессора с физической памятью;
6. реализовать и протестировать модуль.

1.2 Анализ структур ядра USB-подсистемы

Для управления внешним накопителем, таким как USB-устройство в операционной системе представлено несколько структур.

1.2.1 Анализ структуры usb_device

Структура `usb_device` отвечает за представление USB-устройства в ядре. В листинге 1.1 представлены основные поля этой структуры [2].

Листинг 1.1 – Определение структуры usb_device (часть 1)

```
struct usb_device {
    int     devnum;
    char        devpath[16];
    enum usb_device_state   state;
    ...
    struct usb_device_descriptor descriptor;
    ...
    unsigned can_submit:1;
    ...
```

Листинг 1.2 – Определение структуры usb_device (часть 2)

```
    char *product;
    char *manufacturer;
    char *serial;
    ...
};
```

Анализ полей структуры usb_device

— `devnum` — номер устройства в системе. Уникальный идентификатор, присваиваемый устройству при его подключении;

— `devpath[16]` — путь к устройству в дереве USB. Строка, представляющая путь от корневого хаба до устройства;

— `state` — текущее состояние устройства;

— `descriptor` — содержит стандартный дескриптор устройства (идентификатор производителя, идентификатор продукта, версия устройства и т.д.);

— `can_submit` — флаг, указывающий, может ли устройство отправлять URBs;

— `product` — название или модель самого устройства;

— `manufacturer` — название компании-производителя устройства;

— `serial` — уникальный строковый идентификатор конкретного экземпляра устройства.

1.2.2 Анализ структуры urb

URB(User Request Block) инкапсулирует запрос на передачу данных, передаваемый драйвером USB-устройства низкоуровневому драйверу хост-контроллера. В листинге 1.3 представлены основные поля этой структуры [2].

Листинг 1.3 – Определение структуры URB

```
struct urb {
    ...
    struct usb_device *dev;
    ...
    int status;
    unsigned int transfer_flags;
    void *transfer_buffer;
    dma_addr_t transfer_dma;
    struct scatterlist *sg;
    int num_sgs;
    u32 transfer_buffer_length;
    u32 actual_length;
    ...
    usb_complete_t complete;
};
```

Анализ полей структуры URB

— `dev` — представление устройства USB;

— `status` — статус выполнения;

— `transfer_flags` — флаги передачи. Флаг URB_NO_TRANSFER_DMA_MAP сообщает драйверу хост-контроллера, что драйвер устройства поддерживает DMA(Direct Memory Access);

— `transfer_buffer` — указатель на область данных для передачи;

— `transfer_dma` — физический адрес передаваемых данных. Если установлен флаг URB_NO_TRANSFER_DMA_MAP драйвер устройства сообщает, что он предоставил этот адрес DMA, который драйвер хост-контроллера должен использовать предпочтительнее, чем transfer_buffer;

— `sg` — указатель на начало списка разрозненных буферов для передачи;

— `num_sgs` — длина списка разрозненных буферов;

— `transfer_buffer_length` — размер передаваемого буфера;

— `actual_length` — фактически переданное количество байт;

— `complete` — функция обратного вызова. Используется для освобождения или повторной отправки URB.

Типы передач, поддерживаемые URB

URB поддерживает все четыре типа передач, определенные в спецификации USB [3]:

1. управляющие посылки (Control Transfers), используемые для конфигурирования во время подключения и в процессе работы для управления устройствами.

2. сплошные передачи (Bulk Data Transfers) сравнительно больших пакетов без жестких требований ко времени доставки. Приоритет этих передач самый низкий, они могут приостанавливаться при большой загрузке шины.

3. прерывания (Interrupt) — короткие передачи, имеют спонтанный характер и должны обслуживаться не медленнее, чем того требует устройство.

4. изохронные передачи (Isochronous Transfers) — непрерывные передачи в реальном времени, занимающие предварительно согласованную часть пропускной способности шины и имеющие заданную задержку доставки.

Каждая операция ввода-вывода через USB шину оформляется в виде отдельного URB, что обеспечивает единый интерфейс для работы с различными типами USB-передач.

1.3 Анализ функций ядра для работы с USB

Для работы с устройствами USB в ядре существует множество функций, описанных в файлах `/drivers/usb/core/urb.c`, `/drivers/usb/core/usb.c`, `/drivers/usb/core/hcd.c`.

1.3.1 Анализ функции usb_submit_urb

Функция отправляет запрос на передачу, передавая URB, описывающее этот запрос, в подсистему USB. Завершение запроса идентифицируется

позднее, асинхронно, при вызове обработчики завершения. В листинге 1.4 представлены основные этапы выполнения функции [2].

Листинг 1.4 – Основные этапы выполнения функции usb_submit_urb

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
{
    int xfertype;
    struct usb_device *dev;
    struct usb_host_endpoint *ep;
    dev = urb->dev;
    // определение конечной точки передачи
    ep = usb_pipe_endpoint(dev, urb->pipe);
    if (!ep)
        return -ENOENT;
    urb->ep = ep;
    // установка статуса URB - в процессе выполнения
    urb->status = -EINPROGRESS;
    urb->actual_length = 0;
    // определение типа передачи
    xfertype = usb_endpoint_type(&ep->desc);
    if (xfertype == USB_ENDPOINT_XFER_CONTROL) {
        ...
    }
    // обработка контроллером хоста
    return usb_hcd_submit_urb(urb, mem_flags);
}
```

URBs могут отправляться конечным точкам до завершения предыдущих, чтобы минимизировать задержки. Таким образом очередь на конечной точке никогда не будет пустой. Такая организация позволяет максимально эффективно использовать полосу пропускания, позволяя USB контроллерам начинать работу с более поздними запросами до того, как программное обеспечение драйвера завершит обработку более ранних запросов.

В случае успеха функция возвращает 0, иначе – отрицательный номер ошибки.

1.3.2 Анализ функции usb_hcd_link_urb_to_ep

Функция добавляет URB в очередь конечной точки. Принимает структуру хост-контроллера, на который был отправлен URB и сам URB. В листинге 1.5 представлены основные этапы выполнения функции [2].

Листинг 1.5 – Основные этапы выполнения функции usb_hcd_link_urb_to_ep

```
int usb_hcd_link_urb_to_ep(struct usb_hcd *hcd, struct urb *urb)
{
    int     rc = 0;
    // обеспечение монопольного доступа
    spin_lock(&hcd_urb_list_lock);
    // отменён ли URB
    if (unlikely(atomic_read(&urb->reject))) {
        rc = -EPERM;
        goto done;
    }
    // могут ли URB быть отправлены на эту конечную точку
    if (unlikely(!urb->ep->enabled)) {
        rc = -ENOENT;
        goto done;
    }
    // может ли устройство отправлять запросы
    if (unlikely(!urb->dev->can_submit)) {
        rc = -EHOSTUNREACH;
        goto done;
    }
    // добавление urb в очередь конечной точки
    if (HCD_RH_RUNNING(hcd)) {
        urb->unlinked = 0;
        list_add_tail(&urb->urb_list, &urb->ep->urb_list);
    } else {
        rc = -ESHUTDOWN;
        goto done;
    }
done:
    spin_unlock(&hcd_urb_list_lock);
    return rc;
}
```

Драйверы хост-контроллера должны вызывать эту процедуру в своем методе `enqueue()`. Возвращает значение 0, если ошибки нет, в противном случае возвращается отрицательный код ошибки (в этом случае метод `enqueue()` должен завершиться ошибкой).

1.4 Анализ методов перехвата управления у функций ядра

В ядре Linux существует несколько механизмов перехвата управления у функций ядра.

1.4.1 kprobes

Kprobes — это средство динамической отладки ядра, позволяющее ставить точки останова(breakpoints) на доступные для записи участки памяти и самостоятельно обрабатывать их. Описывается структурой `kprobe`, имеющей следующие поля:

Листинг 1.6 – Определение структуры kprobe

```
struct kprobe {
    ...
    kprobe_opcode_t *addr;
    const char *symbol_name;
    ...
    kprobe_pre_handler_t pre_handler;
    kprobe_post_handler_t post_handler;
    ...
};
```

— `symbol_name` — имя функции, на которую ставится breakpoint;

— `pre_handler` — указатель на функцию обработки точки останова до выполнения основной функции;

— `post_handler` — указатель на функцию обработки точки останова после выполнения основной функции;

— `addr` — адрес функции, на которую ставится breakpoint. Должно быть указано либо имя функции, либо её адрес, если заполнены оба поля, то регистрация kprobe вернёт ошибку -EINVAL [4].

Добавление и удаление точек останова выполняют следующие функции:

Листинг 1.7 – Функции добавления и удаления kprobes

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *kp);
int register_kretprobe(struct kretprobe *rp);
void unregister_kprobe(struct kprobe *kp);
void unregister_kretprobe(struct kretprobe *rp);
```

1.4.2 Livepatch

Livepatch (Kernel Live Patching) — механизм для динамического обновления работающего ядра без перезагрузки системы. Он позволяет заменять функции ядра на новые версии "на лету". Хотя изначально предназначен для исправления уязвимостей, его можно адаптировать для перехвата вызовов функций путём замены оригинальных функций на модифицированные.

Структура `klp_func` описывает подменяемую функцию:

Листинг 1.8 – Определение структуры klp_func

```
struct klp_func {
    const char *old_name;
    void *new_func;
    ...
    void *old_func;
    ...
    struct list_head stack_node;
    ...
    bool patched;
    ...
};
```

— `old_name` — имя оригинальной функции;

— `new_func` — указатель на исправленную функцию;

— `old_func` — указатель на заменяемую функцию;

— `stack_node` — узел списка для отслеживания вызовов функции;

— `patched` — флаг, указывающий, применён ли патч к данной функции.

Структура `klp_object` описывает модуль, содержащий функции для замены:

Листинг 1.9 – Определение структуры klp_object

```
struct klp_object {
    const char *name;
    struct klp_func *funcs;
    ...
    struct module *mod;
    ...
    bool patched;
};
```

— `name` — имя модуля;

— `funcs` — список функций для патчинга;

— `mod` — указатель на структуру модуля;

— `patched` — флаг, указывающий, применён ли патч ко всем функциям объекта.

Структура `klp_patch` описывает единый патч, который может включать несколько объектов:

Листинг 1.10 – Определение структуры klp_patch

```
struct klp_patch {
    struct module *mod;
    struct klp_object *objs;
    ...
    bool replace;
    struct list_head list;
    ...
    bool enabled;
    ...
};
```

— `mod` — указатель на модуль, реализующий патч;

— `objs` — объекты, входящие в патч;

— `replace` — флаг, указывающий, должен ли патч заменять предыдущие патчи;

— `list` — узел для включения в глобальный список патчей;

— `enabled` — флаг активности патча.

Для включения патча необходимо вызвать следующую функцию:

Листинг 1.11 – Основные этапы выполнения функции klp_enable_patch

```
int klp_enable_patch(struct klp_patch *patch)
{
    int ret;
    // Проверка, что модуль является livepatch
    if (!is_livepatch_module(patch->mod))
        return -EINVAL;
    mutex_lock(&klp_mutex);
    // Проверка совместимости с другими патчами
    if (!klp_is_patch_compatible(patch)) {
        mutex_unlock(&klp_mutex);
        return -EINVAL;
    }
    // Инициализация патча
    ret = klp_init_patch(patch);
    if (ret)
        goto err;
    // Включение патча
    ret = __klp_enable_patch(patch);
    if (ret)
        goto err;
    mutex_unlock(&klp_mutex);
    return 0;
err:
    mutex_unlock(&klp_mutex);
    return ret;
}
```

1.4.3 Выбор метода перехвата управления у функций ядра

Для выбора оптимального метода перехвата управления у функций были определены следующие критерии:

1. необходимость полной замены функции;
2. количество структур, которые надо инициализировать;
3. минимальная версия ядра;
4. есть ли прямой доступ к аргументам функции (без использования регистров).

В таблице 1.1 представлено сравнение рассмотренных методов.

Таблица 1.1 – Сравнение методов перехвата управления

Метод	1	2	3	4
kprobes	нет	1	2.6.9	нет
Livepatch	да	3	4.0	да

На основании анализа таблицы для реализации в модуле ядра выбран механизм kprobes, так как он:

— проще и быстрее в реализации;

— поддерживается большим количеством версий ядра;

— не требует полной переписи оригинальной функции.

1.5 Анализ методов подмены информации

В данном разделе рассматриваются три подхода: побайтовая замена на константу, заполнение случайными данными и метод Гутмана.

1.5.1 Побайтовая замена на константу

Наиболее простым методом уничтожения информации является замена исходных данных фиксированным значением (чаще всего нулями) с помощью функции `memset()`. Данный подход реализуется минимальными вычислительными затратами, однако обладает существенным недостатком — при последующем чтении данных файлов с флешки будет очевидно, что содержимое файла было изменено сторонним модулем.

1.5.2 Заполнение случайными данными

Более надёжным методом является перезапись данных криптографически стойкими случайными значениями с помощью функции ядра Linux `get_random_bytes()`. Данная функция использует криптографически стойкий генератор псевдослучайных чисел (CSPRNG), что обеспечивает непредсказуемость записываемых данных [5].

1.5.3 Метод Гутмана

Метод Гутмана предполагает 35 циклов перезаписи специально подобранными паттернами, разработанными для преодоления особенностей работы накопителей на магнитных дисках. Для современных твердотельных накопителей (SSD) и флеш-памяти этот метод является избыточным, так как контроллеры таких устройств используют выравнивание износа и другие механизмы, которые делают многократную перезапись неэффективной [6].

1.5.4 Выбор метода подмены информации

Для выбора оптимального метода подмены информации были определены следующие критерии:

1. количество проходов записи — число циклов перезаписи данных;
2. объем дополнительной памяти — память для хранения паттернов или буферов;
3. наличие готовой реализации в ядре Linux;
4. криптографическая стойкость — использование криптографически стойкого генератора случайных чисел;
5. применимость для современных USB-накопителей.

В таблице 1.2 представлено сравнение рассмотренных методов.

Таблица 1.2 – Сравнение методов подмены информации

Метод	1	2	3	4	5
Замена нулями	1	0 байт	да	нет	нет
Метод Гутмана	35	35 байт	нет	нет	нет
Случайные данные	1	0 байт	да	да	да

На основании анализа таблицы для реализации в модуле ядра выбран метод заполнения случайными данными через `get_random_bytes()`, так как он:

— требует минимального количества проходов;

— не использует дополнительную память;

— реализован в ядре Linux;

— обеспечивает криптографическую стойкость через CSPRNG;

— эффективен для современных флеш-накопителей.

1.6 Файловая система FAT32

На рисунке 1.1 изображена структура раздела FAT [7]:

Рисунок 1.1 – Структура раздела FAT32

В файловой системе FAT дисковое пространство логического раздела делится на две области – системную и область данных. Системная область создается и инициализируется при форматировании, а впоследствии обновляется при манипулировании файловой структурой.

В FAT32 корневой каталог может быть расположен в любом месте области данных раздела и иметь произвольный размер. Структура элемента каталога файлов показана на рисунке 1.2. Элемент начинается с 11-байтного поля, содержащего «короткое имя» файла, по которому операционная система обычно осуществляет поиск файла в каталоге. «Короткое имя» состоит из двух полей: 8-байтного поля, содержащего собственно имя файла и 3-байтного поля, содержащего расширение. Разделительная точка между именем и расширением файла не хранится в структуре данных.

Смещение	Размер (байт)	Содержание
0x00	11	Короткое имя файла.
0x0B	1	Атрибуты файла.
0x0C	1	Зарезервировано для Windows NT. Поле обрабатывается только в FAT32.
0x0D	1	Поле, уточняющее время создания файла (содержит десятки миллисекунд). Поле обрабатывается только в FAT32.
0x0E	1	Время создания файла. Поле обрабатывается только в FAT32.
0x10	2	Дата создания файла. Поле обрабатывается только в FAT32.
0x12	2	Дата последнего обращения к файлу для записи или считывания данных. Поле обрабатывается только в FAT32.
0x14	2	Старшее слово номера первого кластера файла. Поле обрабатывается только в FAT32.
0x16	2	Время выполнения последней операции записи в файл.
0x18	2	Дата выполнения последней операции записи в файл.
0x1A	2	Младшее слово номера первого кластера файла.
0x1C	4	Размер файла в байтах.

Рисунок 1.2 – Структура элемента каталога

В ядре Linux элемент каталога представлен структурой `msdos_dir_entry`:

Листинг 1.12 – Определение структуры msdos_dir_entry

```
struct msdos_dir_entry {
    __u8    name[MSDOS_NAME];/* name and extension */
    __u8    attr;       /* attribute bits */
    __u8    lcase;      /* Case for base and extension */
    __u8    ctime_cs;   /* Creation time, centiseconds (0-199) */
    __le16  ctime;      /* Creation time */
    __le16  cdate;      /* Creation date */
    __le16  adate;      /* Last access date */
    __le16  starthi;    /* High 16 bits of cluster in FAT32 */
    __le16  time,date,start;/* time, date and first cluster */
    __le32  size;       /* file size (in bytes) */
};
```

4-ый бит в поле attr, равный единице, определяет файл как каталог. Если первый байт поля name равен 0xE5, то элемент каталога свободен и его можно использовать при создании нового файла.

В первом секторе резервной области присутствует структура FSINFO. Эта структура содержит информацию о количестве свободных кластеров на диске и о номере первого свободного кластера в таблице FAT. В Linux описывается структурой `fat_boot_fsinfo`:

Листинг 1.13 – Определение структуры fat_boot_fsinfo

```
struct fat_boot_fsinfo {
    __le32   signature1;    /* 0x41615252L */
    __le32   reserved1[120];    /* Nothing as far as I can tell
       */
    __le32   signature2;    /* 0x61417272L */
    __le32   free_clusters; /* Free cluster count.  -1 if
       unknown */
    __le32   next_cluster;  /* Most recently allocated cluster */
    __le32   reserved2[4];
};
```

Поле `signature1` — признак того, что данные сектор содержит структуру FSINFO. Значение поля определено макросом `#define FAT_FSINFO_SIG1 0x41615252`.

Поля `attr`, `name` в структуре `msdos_dir_entry` и `signature1` в структуре `fat_boot_fsinfo` позволяют отличать служебные FAT32 данные в передаваемых URB от непосредственно содержимого файлов.

1.7 Синхронизация кэша при работе с DMA

При перехвате URB-запросов и модификации передаваемых данных необходимо обеспечить согласованность между кэшем центрального процессора и физической памятью. Драйвер хост-контроллера USB использует прямой доступ к памяти (DMA) для чтения данных непосредственно из физических адресов, минуя кэш процессора.

Чтобы гарантировать, что контроллер получает актуальные модифицированные данные, требуется явная синхронизация кэша. Для этого используется функция `dma_sync_single_for_device`, которая обеспечивает запись изменений из кэша процессора в физическую память перед началом операции

DMA от устройства. Код функции представлен в листинге 1.14

Листинг 1.14 – Код функции dma_sync_single_for_device

```
void dma_sync_single_for_device(struct device *dev, dma_addr_t
   addr,
        size_t size, enum dma_data_direction dir)
{
    // Получение операций DMA для конкретного устройства
    const struct dma_map_ops *ops = get_dma_ops(dev);

    BUG_ON(!valid_dma_direction(dir));
    // прямое отображение
    if (dma_map_direct(dev, ops))
        dma_direct_sync_single_for_device(dev, addr, size, dir);
    // косвенное отображение через операции устройства
    else if (ops->sync_single_for_device)
        ops->sync_single_for_device(dev, addr, size, dir);
    debug_dma_sync_single_for_device(dev, addr, size, dir);
}
```

1.8 Выводы

В результате проведённого анализа были выбраны механизм перехвата управления у функций ядра, метод подмены информации, рассмотрены структуры и функции для работы с USB подсистемой, выделены необходимые поля в структурах FAT32, и рассмотрена синхронизация кэша процессора.

В качестве механизма перехвата управления был выбран kprobes, так как он проще и быстрее в реализации, поддерживается большим количеством версий ядра, не требует полной переписи оригинальной функции.

Для подмены информации был выбран метод заполнения случайными данными через `get_random_bytes()`, так как он не использует дополнительную память, реализован в ядре Linux и обеспечивает криптографическую стойкость через CSPRNG.

Для идентификации служебных FAT32 данных в передаваемых URB были выделены поля `attr`, `name` в структуре `msdos_dir_entry` и `signature1` в структуре `fat_boot_fsinfo`.

Для синхронизации кэша процессора и физической памяти была рассмотрена функция `dma_sync_single_for_device`.

2 Конструкторский раздел

2.1 IDEF0 диаграммы

На рисунках 2.1 и 2.2 представлены IDEF0 диаграммы нулевого и первого уровней.

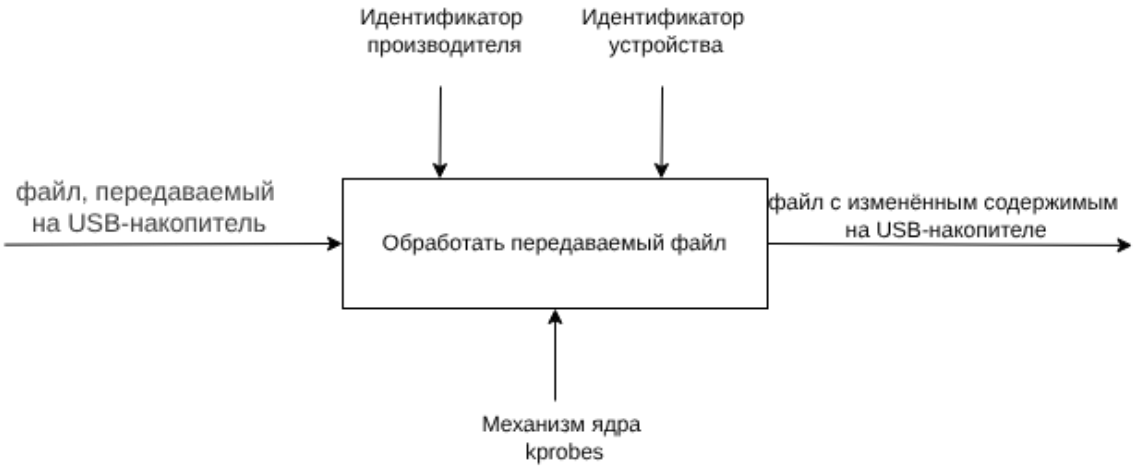

Рисунок 2.1 – Диаграмма IDEF0 нулевого уровня

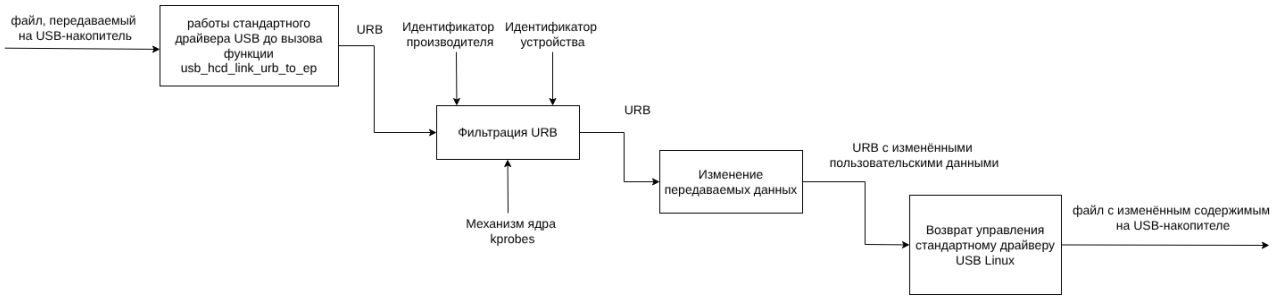

Рисунок 2.2 – Диаграмма IDEF0 первого уровня

2.2 Схема алгоритма фильтрации перехваченных URB

На рис. 2.3 представлена схема работы алгоритма фильтрации перехваченных URB.

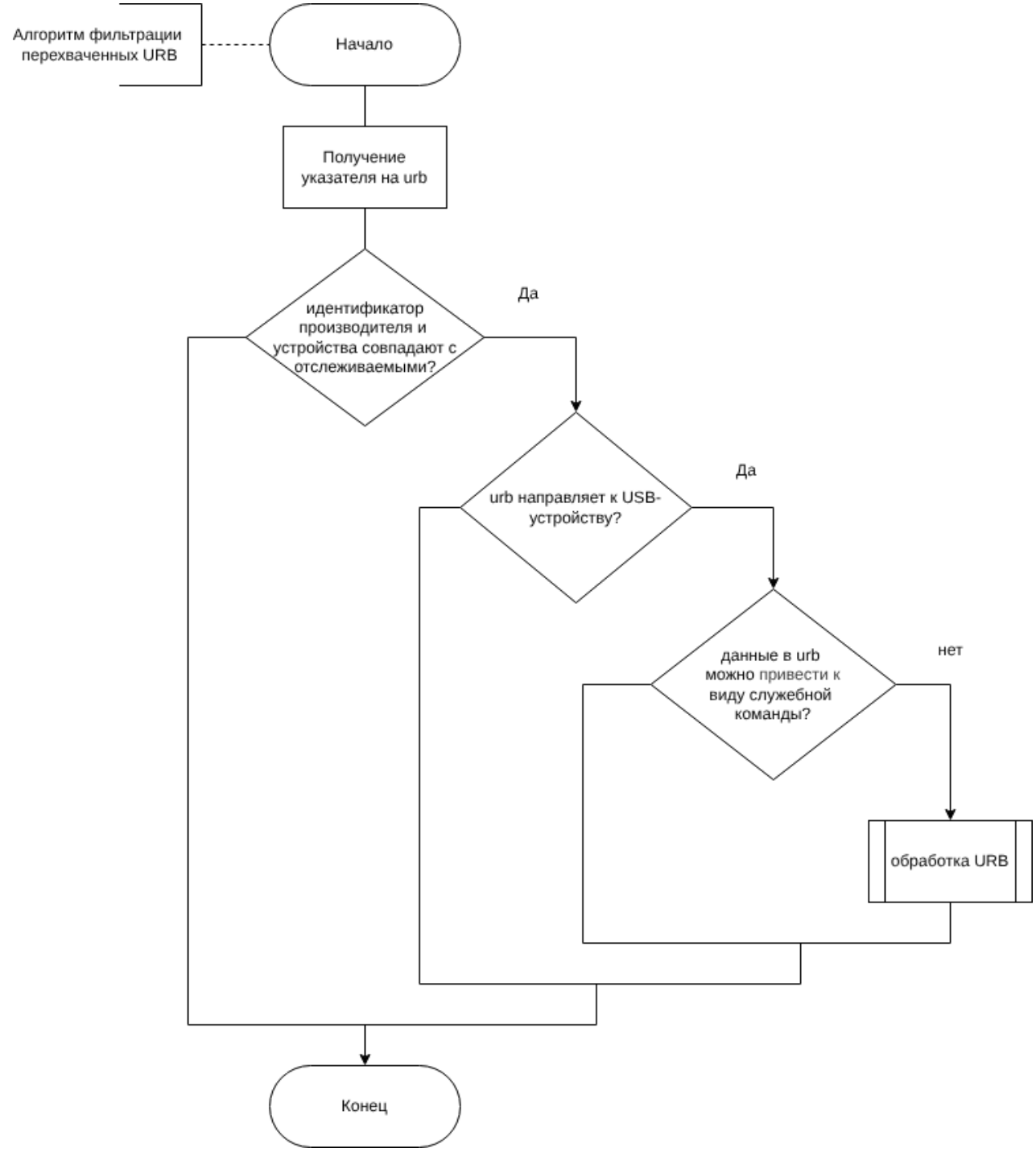

Рисунок 2.3 – Алгоритм фильтрации перехваченных URB

2.3 Схема алгоритма обработки URB

На рис. 2.4 представлена схема работы алгоритма обработки перехваченного URB.

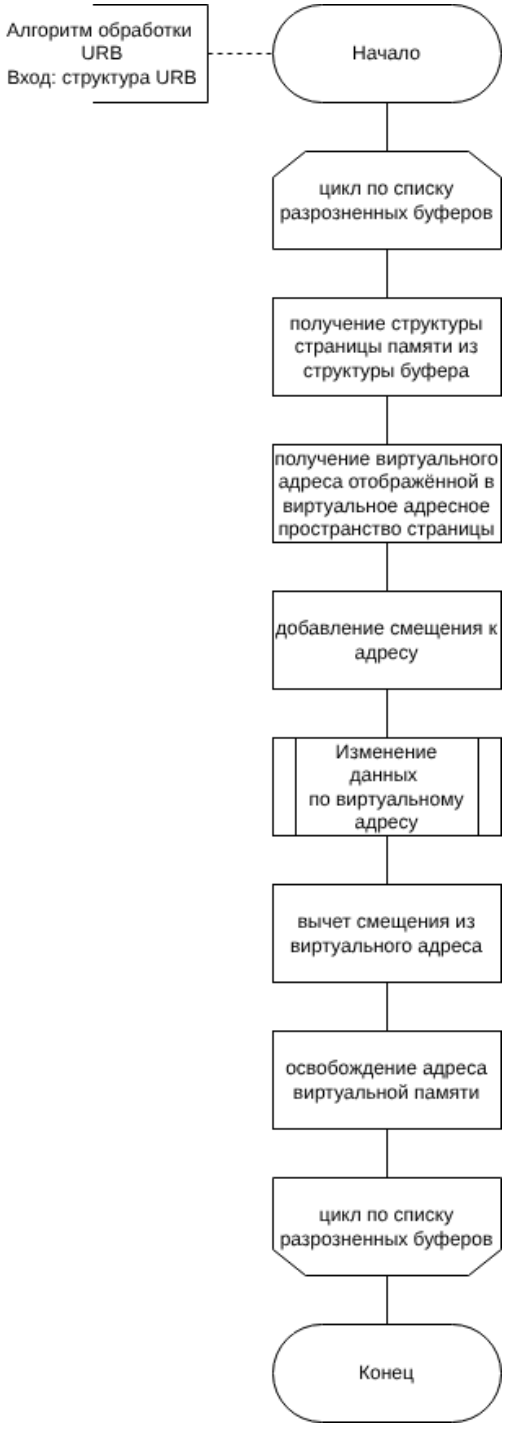

Рисунок 2.4 – Алгоритм обработки перехваченного URB

2.4 Схема алгоритма изменения данных в URB

На рис. 2.5 представлена схема работы алгоритма изменения данных в URB.

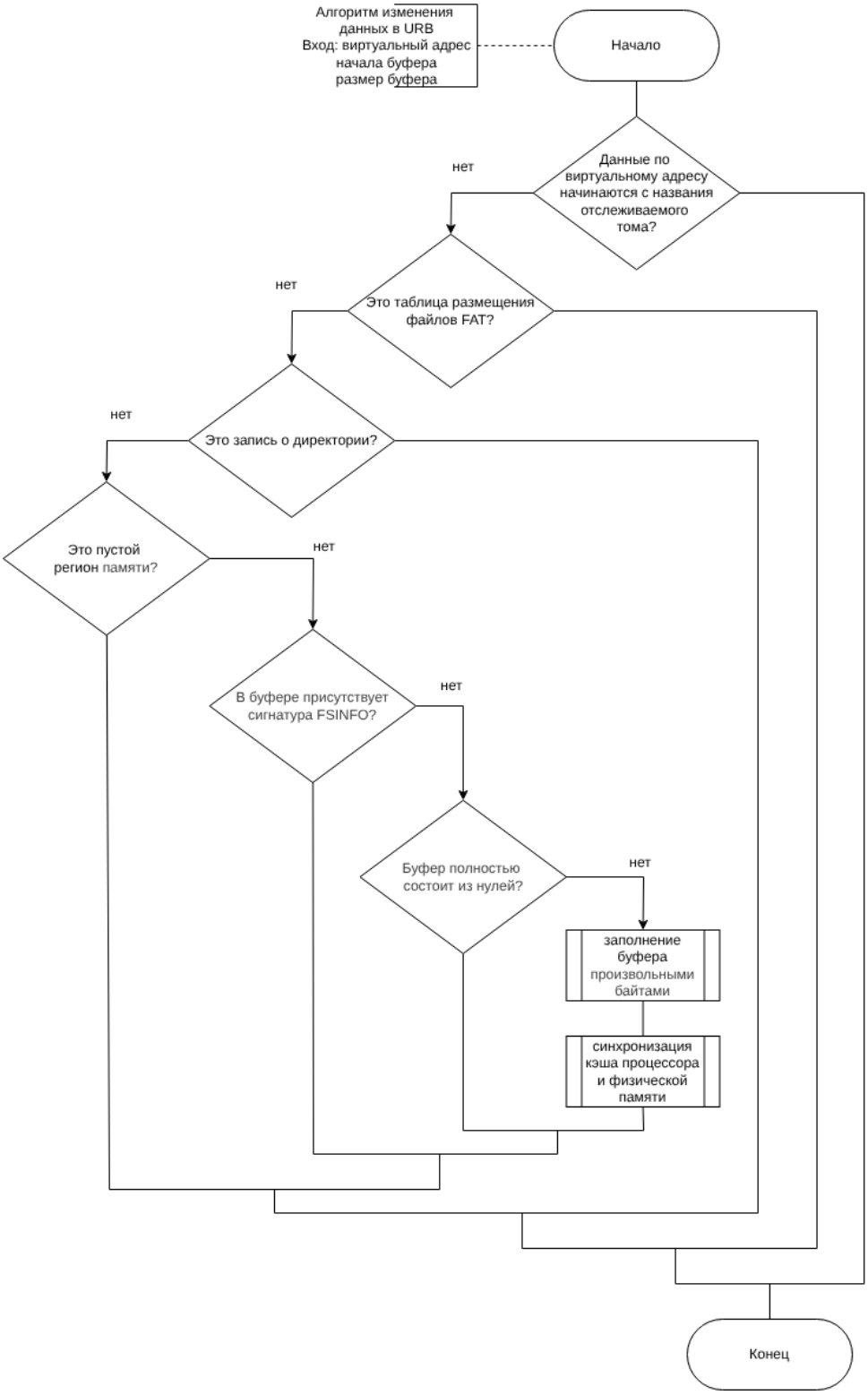

Рисунок 2.5 – Алгоритм изменения данных в URB

3 Технологический раздел

3.1 Выбор языка и среды программирования

Для разработки модуля был выбран язык программирования C, так как на нём написано ядро Linux. Для сборки модуля была использована утилита Make [8]. В качестве среды разработки была использована среда VSCode [9].

3.2 Конфигурация USB-накопителя

На листинге 3.1 представлены параметры отслеживаемого флеш накопителя, такие как идентификаторы производителя и устройства и имя тома, заданные в программе через макросы.

Листинг 3.1 – Параметры отслеживаемого USB-накопителя

```
#define VENDOR_ID   0x0dd8
#define PRODUCT_ID  0x3801
#define VOLUME_NAME "MYFLASH"
```

3.3 Функция проверки буфера на соответствие таблице FAT

На листинге 3.2 представлен код функции, проверяющей, что буфер является таблицей размещения файлов.

Листинг 3.2 – Функция проверки буфера на соответствие таблице FAT

```
static int is_fat_table(const unsigned char *buffer)
{
    if (buffer[0] == 0xF8 &&
        buffer[1] == 0xFF &&
        buffer[2] == 0xFF &&
        buffer[3] == 0x0F &&
        buffer[4] == 0xFF &&
        buffer[5] == 0xFF &&
        buffer[6] == 0xFF &&
        buffer[7] == 0x0F)
            return 1;
    return 0;
}
```

3.4 Функция проверки буфера на заполненность нулями

На листинге 3.3 представлен код функции, проверяющей, что буфер URB состоит из нулей.

Листинг 3.3 – Функция проверки буфера на заполненность нулями

```
static ssize_t find_first_nonzero(const unsigned char *buffer,
   size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        if (buffer[i] != 0)
            return i;
    }
    return -1;
}
```

3.5 Функции, вызываемые при загрузке и выгрузке модуля

На листинге 3.4 представлен код функций, выполняющихся при загрузке и выгрузке модуля. При входе регистрируется `kprobe`, при выгрузке удаляется.

Листинг 3.4 – Функции, вызываемые при загрузке и выгрузке модуля

```
static struct kprobe kp_submit;
static int __init monitor_init(void) {
    kp_submit.symbol_name = "usb_hcd_link_urb_to_ep";
    kp_submit.pre_handler = handler_usb_hcd_link_urb_to_ep_pre;
    int ret = register_kprobe(&kp_submit);
    if (ret < 0) {
        printk(KERN_ERR "Failed kprobe: %d\n", ret);
        return ret;
    }
    printk(KERN_INFO "USB FAKER: Loaded\n");
    return 0;
}
static void __exit monitor_exit(void) {
    unregister_kprobe(&kp_submit);
    printk(KERN_INFO "USB FAKER: Unloaded\n");
}
```

3.6 Функция обработки структуры URB

На листинге 3.5 представлен код функции, которая проверяет передаваемые в URB данные и меняет их с помощью XOR в случае, если они являются внутренним содержимым передаваемых на USB-накопитель файлов.

Листинг 3.5 – Функция обработки структуры URB

```
static void process(struct urb *urb){
    struct scatterlist *sg;
    int i;
    for_each_sg(urb->sg, sg, urb->num_sgs, i) {
        void *virt_addr;
        size_t len = sg->length;
        struct page *page = sg_page(sg);
        if (page) {
            virt_addr = kmap_local_page(page);
            if (virt_addr) {
                virt_addr += sg->offset;
                if (memcmp(virt_addr, VOLUME_NAME,
                   strlen(VOLUME_NAME)) == 0 ||
                    is_fat_table(virt_addr) ||
                    ((char*)virt_addr)[11] == 0x10 ||
                    ((char*)virt_addr)[0] == 0xE5 ||
                    *(__le32 *)&((char*)virt_addr)[0] ==
                       cpu_to_le32(FAT_FSINFO_SIG1) ||
                    find_first_nonzero(virt_addr, len) == -1)
                    goto unmap;
                get_random_bytes(virt_addr, len);
                if (urb->dev->bus && urb->dev->bus->sysdev)
                    dma_sync_single_for_device(
                        urb->dev->bus->sysdev,
                        sg_dma_address(sg),
                        len,
                        DMA_TO_DEVICE);
unmap:
                virt_addr -= sg->offset;
                kunmap_local(virt_addr);
            }
        }
    }
}
```

3.7 Функция-обработчик точки останова

На листинге 3.6 представлен код функции, которая выполняется при достижении точки останова. Указатель на структуру URB берётся из регистра si. Проверяются параметры устройства, направление передачи, является ли передаваемый URB блоком команды. После всех проверок вызывается функция обработки URB.

Листинг 3.6 – Функция-обработчик точки останова

```
static int handler_usb_hcd_link_urb_to_ep_pre(struct kprobe *p,
   struct pt_regs *regs)
{
    struct urb *urb = (struct urb *)regs->si;
    if (!urb || !urb->dev)
        return 0;
    struct usb_device *udev = urb->dev;

    if (udev->descriptor.idVendor == VENDOR_ID &&
        udev->descriptor.idProduct == PRODUCT_ID &&
        usb_urb_dir_out(urb) == 1) {
        struct bulk_cb_wrap *cbw = (struct bulk_cb_wrap
           *)urb->transfer_buffer;
        if (cbw &&
            cbw->Signature &&
            le32_to_cpu(cbw->Signature) == US_BULK_CB_SIGN) {
            return 0;
        }
        process(urb);
    }
    return 0;
}
```

3.8 Makefile

Для сборки проекта была использована утилита Make и написан Makefile, код которого представлен на листинге 3.7.

Листинг 3.7 – Код Makefile

```
obj-m += mon.o
mon-objs := monitor.o
KDIR ?= /lib/modules/$(shell uname -r)/build

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

.PHONY: all
```

Выполнение команды make без параметров приведет к сборке файла загружаемого модуля ядра mon.ko. Для загрузки модуля используется команда sudo insmod mon.ko. Для выгрузки модуля используется команда sudo rmmod mon.

4 Исследовательский раздел

4.1 Исследование работы программы

Для проверки работы модуля на USB-накопитель записывали файлы и папки с разным содержимым.

4.1.1 Тест №1 — текстовый файл

На флеш-накопитель был скопирован файл с содержимым, представленным на листинге 4.1:

Листинг 4.1 – Текст передаваемого файла

```
version: '3.8'
services:
  express:
    build: ./express
    container_name: express-app
    environment:
      - DATABASE_NAME=anekdot_test
      - HOST=postgres
      - PORT=5432
      - USER=postgres
      - PASSWORD=password
    depends_on:
      postgres:
        condition: service_healthy
    deploy:
      resources:
        limits:
          cpus: '1'
          memory: 256M
    ports:
      - "3000:3000"
    networks:
      - benchmark
```

На рисунке 4.1 видно, что содержимое файла превратилось в набор произвольных байтов:

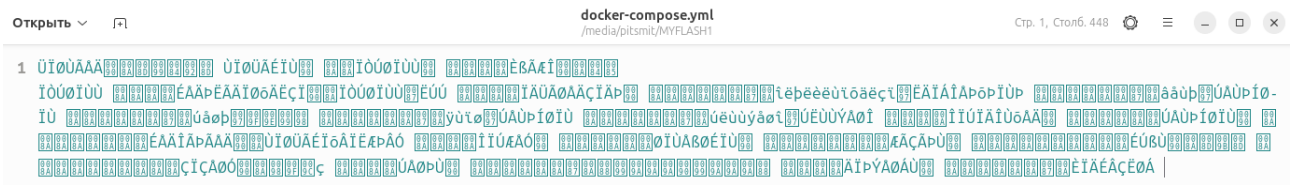

Рисунок 4.1 – Содержимое записанного на USB-накопитель файла

4.1.2 Тест №2 — директория с поддиректориями и файлами

На накопитель была скопирована директория с поддиректориями. В поддиректориях хранились текстовые файлы.

При записи на накопитель структура папок осталась неизменной. На листинге 4.2 представлена структура папок, полученная с помощью команды `tree`.

Листинг 4.2 – Структура переданной папки

```
* MYFLASH1  tree
.
   jest
        config
            jest.config.factory.js
            jest.config.js
        setup
            jest.polyfills.js
            jest.setup.ts

4 directories, 4 files
```

Во всех текстовых файлах данные были изменены, что видно на рисунке 4.2.

Рисунок 4.2 – Содержимое записанного на USB-накопитель файла

4.1.3 Тест №3 — файл PDF

На накопитель был скопирован одностраничный PDF файл. При попытке открыть его с USB-накопителя с помощью стандартного приложения Linux Ubuntu Просмоторщик Документов, было выведено следующее сообщение:

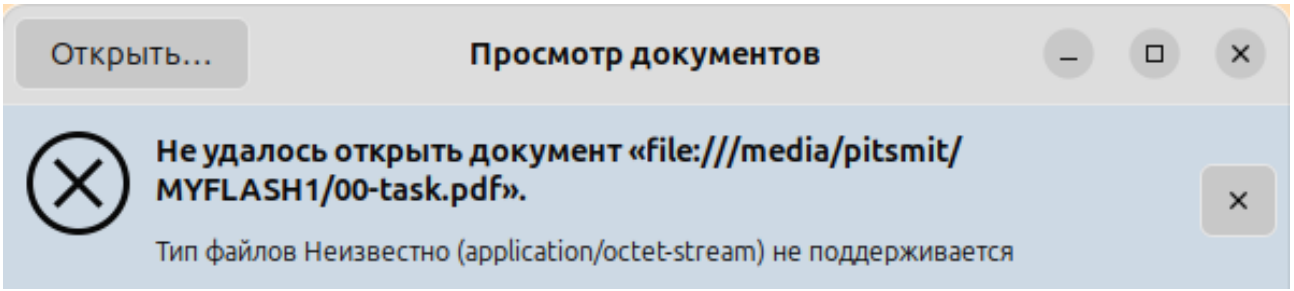

Рисунок 4.3 – Сообщение об ошибке при попытке открыть PDF-файл

В результате записи на флешку была разрушена внутренняя структура файла.

ЗАКЛЮЧЕНИЕ

В ходе работы был разработан загружаемый модуль ядра, подменяющий содержимое передаваемых на флеш-накопитель файлов. Поставленная цель достигнута. Решены все поставленные задачи:

1. проведён анализ функций и структур ядра, представляющих возможность реализации разрабатываемого модуля;
2. проведён анализ методов перехвата управления у функций ядра;
3. проведён анализ методов подмены и повреждения информации;
4. проведён анализ работы файловой системы FAT32;
5. проведён анализ способа синхронизации кэша процессора с физической памятью;
6. реализован и протестирован модуль.

Исследование разработанного программного обеспечения показало, что оно соответствует техническому заданию и решает поставленную задачу: подменяет данные в записываемых на USB-накопитель файлах.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. DLP: предотвращаем утечки; [Электронный ресурс]. — Режим доступа: https://habr.com/ru/companies/otus/articles/798787/ (дата обращения: 03.12.2025).
2. Код ядра Linux; [Электронный ресурс]. — Режим доступа: https://elixir.bootlin.com/linux/v6.12.6/source (дата обращения: 03.12.2025).
3. Шина USB и FireWire; [Электронный ресурс]. — Режим доступа: https://cxem.net/comp/comp56.php (дата обращения: 03.12.2025).
4. Kernel Probes (Kprobes); [Электронный ресурс]. — Режим доступа: https://docs.kernel.org/trace/kprobes.html (дата обращения: 03.12.2025).
5. Secure Random Generators (CSPRNG); [Электронный ресурс]. — Режим доступа: https://cryptobook.nakov.com/secure-random-generators/secure-random-generators-csprng (дата обращения: 03.12.2025).
6. Disk wiping and data forensics: Separating myth from science; [Электронный ресурс]. — Режим доступа: https://www.techrepublic.com/article/disk-wiping-and-data-forensics-separating-myth-from-science/ (дата обращения: 03.12.2025).
7. Мешков В. Архитектура файловой системы FAT //Системный администратор. – 2004. – №. 2. – С. 42-54.
8. Документация Make; [Электронный ресурс]. — Режим доступа: https://www.gnu.org/software/make/manual/make.html (дата обращения: 03.12.2025).
9. Документация VSCode; [Электронный ресурс]. — Режим доступа: https://code.visualstudio.com/ (дата обращения: 03.12.2025).

ПРИЛОЖЕНИЕ А

Листинг 4.3 – Полный код модуля (часть 1)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kprobes.h>
#include <linux/usb.h>
#include <linux/scatterlist.h>
#include <linux/dma-buf.h>
#include <linux/highmem.h>
#include <linux/usb/storage.h>
#include <linux/msdos_fs.h>

#define VENDOR_ID   0x0dd8
#define PRODUCT_ID  0x3801
#define VOLUME_NAME "MYFLASH"
static struct kprobe kp_submit;

static ssize_t find_first_nonzero(const unsigned char *buffer,
   size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        if (buffer[i] != 0)
            return i;
    }
    return -1;
}

static int is_fat_table(const unsigned char *buffer)
{
    if (buffer[0] == 0xF8 &&
        buffer[1] == 0xFF &&
        buffer[2] == 0xFF &&
        buffer[3] == 0x0F &&
        buffer[4] == 0xFF &&
        buffer[5] == 0xFF &&
        buffer[6] == 0xFF &&
        buffer[7] == 0x0F) return 1;
    return 0;
}
```

Листинг 4.4 – Полный код модуля (часть 2)

```
static void process(struct urb *urb)
{
    struct scatterlist *sg;
    int i;
    for_each_sg(urb->sg, sg, urb->num_sgs, i)
    {
        void *virt_addr;
        size_t len = sg->length;
        struct page *page = sg_page(sg);
        if (page)
        {
            virt_addr = kmap_local_page(page);
            if (virt_addr)
            {
                virt_addr += sg->offset;
                printk(KERN_INFO "  SG[%d]: len=%zu\n", i, len);
                if (memcmp(virt_addr, VOLUME_NAME,
                   strlen(VOLUME_NAME)) == 0 ||
                    is_fat_table(virt_addr) ||        // fat
                       table?
                    ((char*)virt_addr)[11] == 0x10 || //
                       directory?
                    ((char*)virt_addr)[0] == 0xE5 || // is free
                       element?
                    *(__le32 *)&((char*)virt_addr)[0] ==
                       cpu_to_le32(FAT_FSINFO_SIG1) ||
                    find_first_nonzero(virt_addr, len) == -1)
                    goto unmap;
                print_hex_dump(KERN_INFO, "    Data: ",
                            DUMP_PREFIX_OFFSET, 16, 1,
                            virt_addr, min(len, 512), true);
                get_random_bytes(virt_addr, len);
                if (urb->dev->bus && urb->dev->bus->sysdev) {
                    dma_sync_single_for_device(
                        urb->dev->bus->sysdev,
                        sg_dma_address(sg),
                        len,
                        DMA_TO_DEVICE);
                }
```

Листинг 4.5 – Полный код модуля (часть 3)

```
unmap:
                virt_addr -= sg->offset;
                kunmap_local(virt_addr);
            }
            else
            {
                printk(KERN_INFO "  SG[%d]: kmap failed\n", i);
            }
        }
        else
        {
            printk(KERN_INFO "  SG[%d]: NULL page\n", i);
        }

    }
}

static int handler_usb_hcd_link_urb_to_ep_pre(struct kprobe *p,
   struct pt_regs *regs)
{
    struct urb *urb = (struct urb *)regs->si;
    if (!urb || !urb->dev)
        return 0;
    struct usb_device *udev = urb->dev;

    if (udev->descriptor.idVendor == VENDOR_ID &&
        udev->descriptor.idProduct == PRODUCT_ID &&
        usb_urb_dir_out(urb) == 1)
        {

        int is_flag = 0;
        if (urb->transfer_flags & URB_NO_TRANSFER_DMA_MAP)
        {
            is_flag = 1;
        }

        printk(KERN_INFO "%d, %d, %d, %s\n", is_flag,
           urb->transfer_buffer_length, urb->num_sgs, (char*)
           urb->transfer_buffer);
```

Листинг 4.6 – Полный код модуля (часть 4)

```
        struct bulk_cb_wrap *cbw = (struct bulk_cb_wrap
           *)urb->transfer_buffer;
        if (cbw &&
            cbw->Signature &&
            le32_to_cpu(cbw->Signature) == US_BULK_CB_SIGN)
        {
            return 0;
        }
        process(urb);
    }
    return 0;
}

static int __init monitor_init(void)
{
    kp_submit.symbol_name = "usb_hcd_link_urb_to_ep";
    kp_submit.pre_handler = handler_usb_hcd_link_urb_to_ep_pre;
    int ret = register_kprobe(&kp_submit);
    if (ret < 0)
    {
        printk(KERN_ERR "Failed kprobe: %d\n", ret);
        return ret;
    }
    printk(KERN_INFO "USB FAKER: Loaded\n");
    return 0;
}

static void __exit monitor_exit(void)
{
    unregister_kprobe(&kp_submit);
    printk(KERN_INFO "USB FAKER: Unloaded\n");
}

module_init(monitor_init);
module_exit(monitor_exit);
MODULE_LICENSE("GPL");
```

Листинг 4.7 – Журнал модуля при выполнении первого теста

```
USB FAKER: Loaded
1, 31, 0, USBC\x1e$
1, 31, 0, USBC\x1f$
1, 31, 0, USBC $
1, 31, 0, USBC!$
1, 31, 0, USBC"$
1, 31, 0, USBC#$
0, 512, 1, (null)
  SG[0]: len=512
Data: 00000000: 76 65 72 73 69 6f 6e 3a 20 27 33 2e 38 27 0a 73
   version: '3.8'.s
Data: 00000010: 65 72 76 69 63 65 73 3a 0a 20 20 65 78 70 72 65
   ervices:.  expre
Data: 00000020: 73 73 3a 0a 20 20 20 20 62 75 69 6c 64 3a 20 2e
   ss:.    build: .
Data: 00000030: 2f 65 78 70 72 65 73 73 0a 20 20 20 20 63 6f 6e
   /express.    con
Data: 00000040: 74 61 69 6e 65 72 5f 6e 61 6d 65 3a 20 65 78 70
   tainer_name: exp
Data: 00000050: 72 65 73 73 2d 61 70 70 0a 20 20 20 20 65 6e 76
   ress-app.    env
Data: 00000060: 69 72 6f 6e 6d 65 6e 74 3a 0a 20 20 20 20 20 20
   ironment:.
Data: 00000070: 2d 20 44 41 54 41 42 41 53 45 5f 4e 41 4d 45 3d
   - DATABASE_NAME=
1, 31, 0, USBC$$
0, 512, 1, (null)
  SG[0]: len=512
1, 31, 0, USBC%$
0, 512, 1, (null)
  SG[0]: len=512
1, 31, 0, USBC&$
0, 512, 1, (null)
  SG[0]: len=512
1, 31, 0, USBC'$
1, 31, 0, USBC($
```