электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

 # 07, июль 2017

УДК 004.451

Доступ к Crypto API ядра ОС Linux из пространства пользователя

***Рыжков С. Е.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Информационные системы и телекоммуникации»

Научный руководитель: Сидякин И.М., к.т.н., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Информационные системы и телекоммуникации»
iu3@bmstu.ru

Введение

В настоящее время при разработке различных систем больше внимание уделяется вопросам безопасности. Особенно это относится к системам, которые работают с конфиденциальной информацией и выполняют различные финансовые операции. К таким системам относятся и различные встраиваемые устройства, например, платежные терминалы. В процессе работы такие устройства должны обеспечивать секретность, целостности и аутентификацию конфиденциальной информации пользователя. Данные свойства можно обеспечить при помощи использования различных криптографических операций, таких как шифрование, хеширование и вычисление кода аутентификации передаваемых данных. Многие стойкие и эффективные алгоритмы уже реализованы и предоставляются разработчиками на коммерческой или бесплатной основе в виде различных криптографических библиотек и пакетов. Производители операционных систем также развивают криптографические возможности своих продуктов и предоставляют разработчикам интерфейсы для использования криптоалгоритмов в драйверах и других компонентах системы. В данной статье будет рассмотрен один из таких интерфейсов – интерфейс Crypto API операционной системы Linux (далее ОС Linux).

ОС Linux широко распространена и популярна среди разработчиков встраиваемых систем за свою доступность для множества платформ, богатые возможности конфигурирования, а также открытость исходных кодов, что позволяет без особого труда модифицировать и собирать полноценную операционную систему под конкретное устройство.

В ядре ОС Linux присутствует собственная реализация многих криптографических алгоритмов, которые предназначены для внутреннего используются в различных компонентах этой системы. Доступ к этим алгоритмам можно получить, используя специальный интерфейс Crypto API, однако данный интерфейс доступен разработчику только в пространстве ядра, т.е. к нему можно обратиться только из кода самого ядра, либо модуля ядра. Каких-либо встроенных средств доступа к Crypto API из пространства пользователя не предусмотрено. Таким образом, если разработчику потребуется использование какого-либо криптографического алгоритма в обычном приложении, которое работает в пространстве пользователя, ему будет необходимо подключать внешние библиотеки, либо реализовывать требуемый алгоритм самостоятельно.

Такой подход вполне приемлем для обычных настольных компьютеров и подобных систем, где одна лишняя библиотека не будет занимать заметное количество памяти. Однако для встраиваемых систем, у которых объем доступной памяти жестко ограничен, размер библиотеки может оказаться критичным. Разработчики таких систем стараются в каждой части системы экономить место. Дублирование реализации криптоалгоритмов в пространстве ядра и пространстве пользователя является нецелесообразным.

Другой важный фактор, который необходимо учитывать - это то, что во встраиваемых системах зачастую применяются специализированные микроконтроллеры и системы на кристалле, которые могут содержать в себе модули аппаратного ускорения криптографических операций. Для доступа к этим ускорителям необходимо задействовать соответствующие драйвера устройств, которые также работают в пространстве ядра. При стандартной реализации драйвера криптографического алгоритма, принятой в системе Linux, его можно зарегистрировать в Crypto API и он будет автоматически задействован при вызове соответствующей криптографической операции. В то время как библиотеки, работающие в пространстве пользователя, не имеют доступа к таким драйверам, а все алгоритмы в них реализованы полностью программно. Т.е. при выполнении операции они задействуют только основные возможности центрального процессора устройства и не обращаются к аппаратным ускорителям.

Использование аппаратных ускорителей может значительно увеличить скорость выполнения криптографических операций, а также сделать их более безопасными, т.к. вся секретная информация не попадает в пространство пользователя, а находится в защищенном адресном пространстве ядра. К тому же, используя аппаратные возможности системы, можно полностью избежать попадания критически важных данных, например,

секретных ключей, в оперативную память, и указать, чтобы они напрямую считывались из защищенной физической памяти в регистры аппаратного криптоускорителя.

Таким образом, в данной статье будет предложен один из возможных способов получения доступа к возможностям Crypto API ОС Linux из пространства пользователя и пример использования данного способа для реализации криптобиблиотеки.

Описание архитектуры

Основная идея предлагаемого подхода состоит в реализации отдельного модуля, который работает в пространстве ядра и имеет прямой доступ к Crypto API. При этом данный модуль должен иметь связь с пространством пользователя, т.е. иметь возможность получить данные и команду из пространства пользователя, обработать ее, вызвав соответствующие функции Crypto API, и вернуть результат обратно.

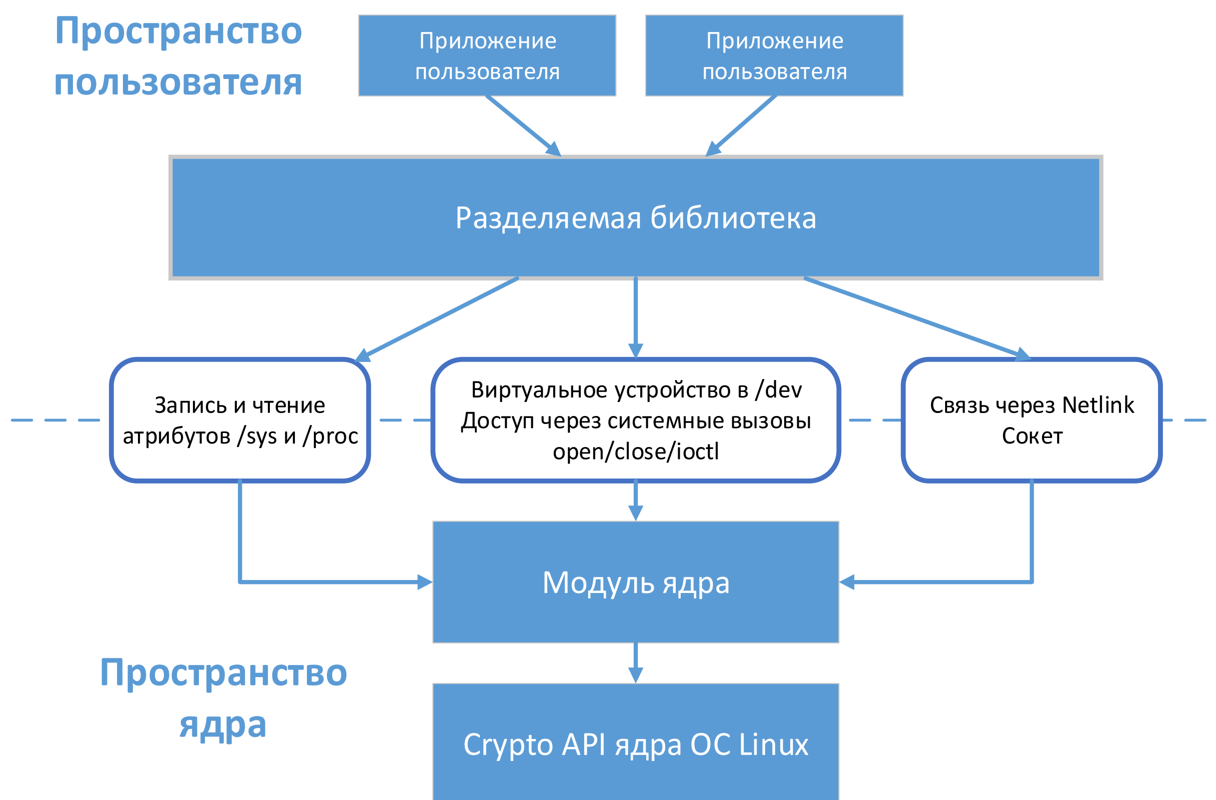

Рис. 1. Обобщенная схема доступа к Crypto API ОС Linux

Существует множество способов связать модуль ядра с пространством пользователя. Основными из них являются: системные вызовы; использование виртуальных файловых систем procfs [1] и sysfs [2]; создание виртуального устройства в каталоге /dev, и выполнение команд посредством системного вызова IOCTL [3]; использование специального типа сокетов – Netlink [4].

Одного модуля ядра было бы достаточно, т.е. приложение пользователя уже может обращаться к CryptoAPI путем записи и чтения атрибутов в файловых системах procfs и sysfs, сокета Netlink, либо путем обращения к виртуальному устройству. Однако, с точки зрения пользовательского приложения работать с таким низкоуровневым интерфейсом не удобно по сравнению с готовыми функциями, которые предоставляют пользователю другие библиотеки.

Чтобы избавить пользователя от прямого обращения к низкоуровневым интерфейсам, в пространстве пользователя можно реализовать небольшую библиотеку с привычным и единообразным интерфейсом для приложений. Таким образом, библиотека скроет внутри себя все детали обращения к модулю ядра за простыми и понятными функциями-обертками. Обобщенная схема доступа к Crypto API ОС Linux из пространства пользователя показана на рис. 1.

Реализация модуля ядра на примере алгоритмов хеширования

Рассмотрим реализацию приведенного выше подхода на примере доступа к функциям хеширования. Модуль ядра можно реализовать в виде драйвера символьного устройства. Функции инициализации и деинициализации модуля могут быть самыми простыми, примеры можно найти в литературе [5]. Всю основную функциональность модуля будут определять функции open, close и ioctl. В данной статье остановимся лишь на ключевых моментах реализации.

```
1  #define IOCTL_INIT_HASH          _IOW(CRYPTOMOD_MAJOR, 1, hash_init_t *)
2  #define IOCTL_UPDATE_HASH        _IOW(CRYPTOMOD_MAJOR, 2, data_update_t *)
3  #define IOCTL_FINALIZE_HASH      _IOR(CRYPTOMOD_MAJOR, 3, char *)
4
5  typedef struct {
6      struct mutex pd_mutex;
7      char *buffer;
8      struct scatterlist *sg;
9      struct hash_desc *desc;
10 } p_data;
11
12 typedef struct {
13     char *alg;
14 } hash_init_t;
15
16 typedef struct {
17     unsigned char *data;
18     unsigned int data_size;
19 } data_update_t;
```

Рис.2. Список команд IOCTL и внутренние структуры модуля

Функция open вызывается при открытии устройства из пространства пользователя. В данной функции нам необходимо подготовить модуль к обработке команд

пользователя, т.е. инициализировать все внутренние приватные данные, которые нужны ему в процессе работы. В случае алгоритмов хеширования внутренние данные – это буфер для чтения данных и связанная с ним структура scatterlist, а также дескриптор алгоритма хеширования hash_desc и мьютекс для предотвращения параллельного доступа к приватным данным (рис. 2). Функция close – это функция, вызываемая при закрытии устройства. Здесь нам необходимо выполнить обратную последовательность действий, т.е. деинициализировать дескриптор hesh_desc, очистить буфер и освободить всю выделенную память.

Функция ioctl является универсальной и может выполнять любые команды, определенные разработчиком, поэтому в данной функции реализуем всю основную логику обмена данными и командами с пространством пользователя.

Работу с алгоритмами хеширования в Crypto API можно разделить на 3 стадии: инициализация алгоритма, обновление данных для подсчета дайджеста и деинициализация операции с выводом результата. Поэтому в функции ioctl реализуем 3 команды: IOCTL_INIT_HASH, IOCTL_UPDATE_HASH, IOCTL_FINALIZE_HASH, пример определения которых представлен на рис. 2. Этим командам будут соответствовать функции init_hash, update_hash и finalize_hash, каждая из которых будет выполнять одну из стадий работы с алгоритмом хешированя. Упрощенные примеры реализации этих функций показаны на рис. 3 и 4.

В функции init_hash происходит инициализация дескриптора алгоритма хеширования. Для этого функция должна получить в параметрах команды строку с названием алгоритма, зарегистрированным в системе. Список всех зарегистрированных алгоритмов можно узнать, прочитав файл /proc/crypto. Для инициализации алгоритма необходимо сначала получить так называемый объект трансформации, вызвав функцию crypto_alloc_hash. Далее необходимо указать в нашем дескрипторе struct hash_desc на созданный объект трансформации и вызвать функцию crypto_hash_init.

```c
static int init_hash(hash_init_t *ih, struct hash_desc **desc)
{
    struct crypto_hash *tfm = crypto_alloc_hash(ih->alg, 0, CRYPTO_ALG_ASYNC);
    if (IS_ERR(tfm))
        return PTR_ERR(tfm);
    *desc = (struct hash_desc *)kzalloc(sizeof(struct hash_desc), GFP_KERNEL);
    if (!(*desc)) {/* Обработка ошибки */}
    (*desc)->tfm = tfm;
    (*desc)->flags = 0;
    if (crypto_hash_init(*desc)) {/* Обработка ошибки */}
    return 0;
}

static int finalize_hash(struct hash_desc *desc, char **out, int *out_size)
{
    *out_size = SHA256_SIZE;
    *out = (char *)kzalloc(SHA256_SIZE, GFP_KERNEL);
    if (!(*out)) {/* Обработка ошибки */}
    if (crypto_hash_final(desc, *out)) {/* Обработка ошибки */}
    return 0;
}
```

Рис 3. Упрощенный пример реализации функций инициализации и деинициализации алгоритма хеширования

В функции update_hash происходит чтение выходных данных из пространства пользователя и обновление значения функции хеширования, в соответствии с этими данными. Для этого необходимо вызвать функцию crypto_hash_update, в параметрах которой передается дескриптор алгоритма хеширования, указатель на буфер данных и размер данных для обновления. При реализации данной функции стоит учитывать тот факт, что устройство может быть открыто параллельно несколькими разными приложениями и одновременно с ними обрабатывать данные. Чтобы избежать чрезмерного выделения памяти при обработки данных большого размера, копирование информации из пространства пользователя необходимо выполнять небольшими порциями через буфер, ограниченного размера, выделенный при открытии устройства.

```c
static int update_hash(struct hash_desc *desc, char *buffer,
                        struct scatterlist *sg, data_update_t *data)
{
    int i;
    unsigned int count = data->data_size / BUFFER_SIZE;
    unsigned int rest = data->data_size % BUFFER_SIZE;
    char *data_ptr = data->data;
    for (i = 0; i < count; i++)
    {
        if (copy_from_user(buffer, data_ptr, BUFFER_SIZE) != 0) {/* Обработка ошибки */
        if (crypto_hash_update(desc, sg, BUFFER_SIZE)) {/* Обработка ошибки */}
        data_ptr += BUFFER_SIZE;
    }
    if (rest > 0)
    {
        if (copy_from_user(buffer, data_ptr, rest) != 0) {/* Обработка ошибки */}
        if (crypto_hash_update(desc, sg, rest)) {/* Обработка ошибки */}
    }
    return 0;
}
```

Рис 4. Упрощенный пример реализации функции обновления данных

В функции finalize_hash происходит возврат подсчитанного значения функции хеширования в пространство пользователя. Для этого вызывается функция crypto_hash_final, в параметрах которой передается дескриптор алгоритма хеширования и указатель на буфер для записи результата.

Пример реализации библиотеки пространства пользователя

Библиотека пространства пользователя представляет собой один простой класс, структура класса представлена на рис. 5. В объекте данного класса достаточно хранить только дескриптор открытого устройства – fd. Методы класса являются функциями-обертками над системными вызовами open, close и ioctl, которые используются для обращения к устройству. Упрощенный пример реализации публичных функций показан на рисунке 6. Открытие и закрытие виртуального устройства можно выполнить соответственно в конструкторе и деструкторе, либо внутри методов init и finalize. В этом случае после получения результата функции хеширования и для обработки новых данных не обязательно удалять объект и создавать новый, а достаточно повторно вызвать цикл функций init, update и finalize.

CryptoAPI
- driver_name : const char*
- fd : int
+ CryptoAPI() «constructor»
+ init(ih : hash_init_t*) : int
+ update(data : unsigned char*, size : int) : int
+ finalize(output : unsigned char*) : int
+ ~ CryptoAPI() «destructor»

Рис 5. Структура класса CryptoAPI библиотеки пространства пользователя

```
1  int CryptoAPI::init(hash_init_t *ih)
2  {
3      fd = open(driver_name, O_RDWR);
4      return ioctl(fd, IOCTL_INIT_HASH, ih);
5  }
6
7  int CryptoAPI::finalize(unsigned char *output)
8  {
9      int retval = ioctl(fd, IOCTL_FINALIZE_HASH, output);
10     close(fd);
11     return retval;
12 }
13
14 int CryptoAPI::update(unsigned char *data, int size)
15 {
16     data_update_t data_t;
17     data_t.data = data;
18     data_t.data_size = size;
19     return ioctl(fd, IOCTL_UPDATE_HASH, &data_t);
20 }
```

Рис 6. Упрощенный пример реализации публичных методов класса CryptoAPI

Тестирование модуля ядра и библиотеки

В первую очередь было проведено функциональное тестирование, чтобы убедиться в работоспособности предложенного подхода. Для тестирования были использованы открытые тестовые вектора алгоритма хеширования SHA-256, которые доступны на сайте NIST [6]. Все тесты прошли успешно.

Для сравнения скорости работы предложенного подхода было решено использовать библиотеку libcrypto, которая входит в состав OpenSSL. OpenSSL содержит высокопроизводительные реализации криптографических алгоритмов и является одной из наиболее успешных и широко используемых криптографических библиотек [7]. Для измерения времени была использована библиотека chrono, которая входит в состав стандартной библиотеки языка Си++. Данная библиотека предоставляет широкие возможности измерения времени с различной степенью точности.

Скорость работы была измерена на тех же тестовых векторах, на которых производилось функциональное тестирование. Результаты сравнения представлены на рис. 7.

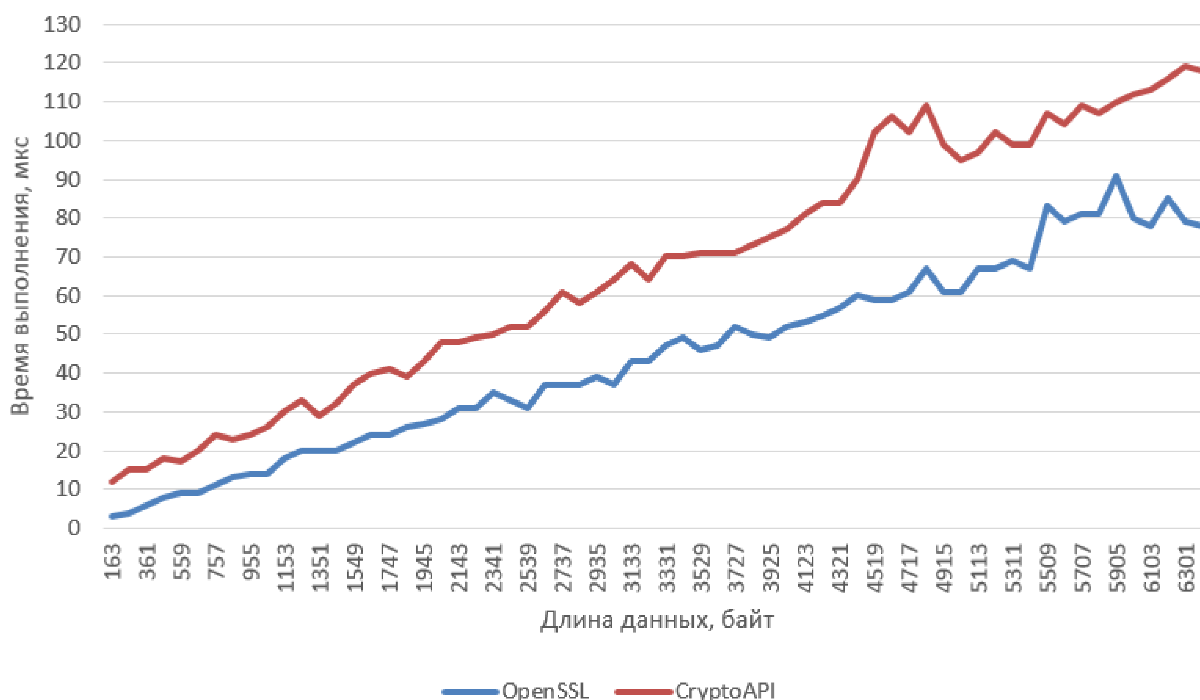

Рис 7. Сравнение скорости работы алгоритма SHA-256 с реализацией в OpenSSL

По представленным графикам мы видим, что подсчет значения алгоритма хеширования SHA-256 при использовании предложенного подхода проходит примерно в 1,5 раза медленнее по сравнению с выполнением аналогичной процедуры, используя

OpenSSL. Однако, следует учитывать тот факт, что тестирование проводилось в ОС Linux, запущенной в виртуальной машине VirtualBox на обычном персональном компьютере с процессором общего назначения и без специализированных аппаратных ускорителей криптоалгоритмов. К тому же библиотека OpenSSL разрабатывается с 1998 года [7], и на протяжении всех этих лет улучшается и оптимизируется опытными разработчиками. Для используемого оборудования такой результат является хорошим и ожидаемым. При этом предполагается, что во встраиваемых системах, которые обладают не высокими показателями производительности центрального процессора в пару сотен мегагерц, но имеют аппаратные модули ускорения криптоалгоритмов, использование данного подхода даст ощутимый прирост скорости выполнения криптографической операции и будет в целом быстрее, чем выполнение аналогичной операции только на центральном процессоре устройства.

Помимо скорости, также не стоит забывать о таком факторе, как уменьшении объема занимаемого в памяти пространства и об отсутствии дублирования алгоритмов в пространстве ядра и пространстве пользователя, что может оказаться решающим при выборе между предлагаемым подходом и обычной реализации с использованием дополнительной библиотеки. Например, на тестовом компьютере, размер модуля ядра составил 6,9 кб, а размер связанной с ним библиотеки – 7,4 кб, т.е. общий объем занимаемого пространства – 14,3 кб при условии, что мы не учитываем размер реализации криптоалгоритмов в самом Crypto API, т.к. они в любом случае будут входить в состав ядра, даже при использовании внешних библиотек. При этом размер библиотеки libcrypto на той же платформе составляет 1.7 Мб. Конечно, можно скомпилировать библиотеку с исключением из нее ненужных нам алгоритмов. Тем не менее размер данной библиотеки будет больше нашей реализации, потому что при добавлении новых алгоритмов в нашем классе CryptoAPI нам нужно будет добавить лишь дополнительные команды для системного вызова ioctl и новые параметры инициализации (например, для алгоритмов шифрования, дополнительным параметром инициализации, помимо названия алгоритма, будет секретный ключ шифрования). Таким образом, при использовании данного подхода размер нашей библиотеки в пространстве пользователя практически не будет расти.

Тут же можно выделить и недостаток предложенного подхода – запрашиваемый алгоритм должен обязательно быть реализован в ядре и зарегистрирован с Crypto API. При его отсутствии пользователю придется самому его реализовывать, либо использовать дополнительные библиотеки. Однако в настоящее время в ядре ОС Linux реализованы все наиболее популярные и распространенные алгоритмы, такие как AES, DES, Triple DES,

RSA, DSA, MD, SHA, HMAC и другие, что покрывает потребности большинства пользователей.

Заключение

В данной работе были рассмотрены возможные варианты доступа к Crypto API ядра ОС Linux из пространства пользователя, а также был предложен подход к реализации криптобиблиотеки, использующей данный интерфейс и алгоритмы, реализованные в ядре. Основным преимуществом такого подхода является отсутствие дублирования алгоритмов в пространстве пользователя, и, как следствие, уменьшение до минимума размера криптобиблиотеки, а также возможность прямого использования аппаратных ускорителей для повышения скорости выполнения операций.

На примере алгоритмов хеширования был представлен вариант реализации описанного подхода, использующий для доступа к Crypto API виртуальное символьное устройство в каталоге /dev. Тестирование данной реализации прошло успешно и показало возможность применения данного подхода. Также было проведено сравнение скорости выполнения операций с популярным криптографическим пакетом OpenSSL на платформе x86.

В дальнейшем планируется провести тестирование производительности на других платформах с малой тактовой частотой центрального процессора, но с наличием аппаратных модулей ускорения криптографических алгоритмов. Помимо этого, планируется дальнейшее развитие модуля ядра и криптобиблиотеки путем добавления поддержки различных типов алгоритмов. Также можно расширять ее функциональность криптобиблиотеки, добавляя возможности, отсутствующие в других подобных библиотеках пространства пользователя, например, хранение ключей в защищенной памяти и доступ к ним по индексу, а также очистка контекста операции по таймауту с использованием аппаратного генератора случайных числе.

Список литературы

[1]. Terrehon Bowden, Bodo Bauer, Jorge Nerin, Shen Feng. The Proc Filesystem. Available at: https://www.kernel.org/doc/Documentation/filesystems/proc.txt, accessed 15.05.2017.

[2]. Patrick Mochel, Mike Murphy. Sysfs - The filesystem for exporting kernel objects. Available at: https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt, accessed 15.05.2017.

[3]. Robert Love. Linux System Programming, Second Edition. O'Reilly Media, Inc., 2013. 456 p.

[4]. RFC 3549. Linux Netlink as an IP Srvices Protocol. Available at: https://tools.ietf.org/html/rfc3549, accessed 15.05.2017.

[5]. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux Device Drivers, 3rd Edition. O'Reilly Media, Inc., 2005. 640 p.

[6]. NIST - Cryptographic Algorithm Validation Program (CAVP). Available at: http://csrc.nist.gov/groups/STM/cavp/secure-hashing.html#sha-2, accessed 17.05.2017.

[7]. Ivan Ristic. OpenSSL Cookbook. A Guide to the Most Frequently Used OpenSSL Features and Commands. London, Feisty Duck Limited, 2013. 49 p.