Расширение и использование Linux Crypto API

[0] Интро

Криптографический *API* в *Linux* введён с версии 2.5.45 ядра. С тех пор *Crypto API* оброс всеми популярными (и не только) международными стандартами:

- симметричного шифрования: *AES*, *Blowfish*, ...
- хэширования: *SHA1/256/512*, *MD5*, ...
- имитозащиты: *hMAC-SHA1/256/512*, *hMAC-MD5*, ...
- AEAD: *AES-GCM*, *ChaCha20-Poly1305*, ...
- генерации псевдослучайных чисел: *FIPS*, ...
- асимметричного шифрования: *RSA*
- согласования ключей: *DH*, *ECDH*
- сжатия: *Deflate*, *ZLIB*

Эта криптография доступна и, в основном, используется различными подсистемами ядра (в *kernel space*): *IPsec*, *dm-crypt* и др. Воспользоваться функциями *Crypto API* из пространства пользователя (*user space*) также возможно через *Netlink*-интерфейс, в который, начиная с версии 2.6.38 ядра, введено семейство *AF_ALG*, предоставляющее доступ к криптографии ядра из *user space* кода. Однако, существующего функционала иногда недостаточно, таким образом появляется необходимость расширения *Crypto API* новыми алгоритмами.

Так, примерно два года назад передо мной стояла задача встраивания национальной (в Республике Беларусь) криптографии в *IPsec*-реализацию ядра *Linux* (версии 3.13). На тот момент я никогда не занимался программированием в *kernel space* и это было моим первым опытом написания модулей ядра, с чем мне очень помогла книга **Роберта Лава "Ядро Linux. Описание процесса разработки"**. Куда сложнее было разбираться с самим *Crypto API* - в упомянутой книге этот вопрос не освещается в принципе, а в официальной документации, хоть и дано достаточно подробное описание архитектуры, практически нет информации о внедрении новых алгоритмов. В своих поисках, в журнале "Хакер" я наткнулся на замечательную статью, популярно рассказывающую о том, *как использовать Crypto API*, но, опять же, там нет ни слова о его расширении. В конце концов, за необходимыми мне знаниями я обратился к первоисточнику (к коду ядра), и решил поставленную задачу.

С тех пор уже прошло некторое время и только сейчас, актуализировав свои знания, я взялся за написание этого текста. Он о *Crypto API*. На конкретных примерах мы посмотрим, каким образом в *Crypto API* можно встроить произвольные алгоритмы и как к ним обратиться как из *kernel space*, так и из *user space*. Я уверен, что этот материал будет крайне полезен разработчикам, столкнувшимся со схожими проблемами, но надеюсь, что и случайный читатель сможет найти здесь для себя что-то интересное. Добро пожаловать под кат!

P.S. Компилируемые исходники моих программ, фрагменты которых будут встречаться ниже, доступны на *github*. Там же - исходник этой статьи. Весь код был написан и протестирован под десктопной Ubuntu 16.04 LTS с ядром версии 4.13.0-32-generic. Некоторые интерфейсы, которые я использовал в модулях, были введены в 4 версии ядра, поэтому на ядрах 3.* они точно *НЕ* скомпилируются.

[1] Архитектура Crypto API

Прежде чем приступить к написанию кода, поговорим немного о том, как всё устроено в *Crypto API*. Впрочем, до меня об этом подробно рассказали в документации, я же, в свою очередь, приведу лишь выжимку, которой должно быть достаточно для понимания материала. Сразу оговорюсь, далее речь пойдёт преемущественно о симметричном шифровании, иные виды криптопреобразований, в целом, работают схожим образом, но многочисленные нюансы очень сильно бы всё усложнили.

Итак, из введения мы уже знаем, что криптографическая подсистема ядра включает множество реализаций криптографических алгоритмов и предоставляет к ним интерфейсы, доступные пользователю (здесь и далее - под пользователем и разработчиком имеются ввиду, соответственно, *пользователь API* и *разработчик API*). В терминах *Crypto API*, алгоритмы называются "преобразованиями" (*transformations*), так, различные типы преобразований имеют собственный под-*API*, а дескрипторам (хэндлам) преобразований, как правило, присваивается имя "*tfm*", например:

```
struct crypto_cipher *tfm = crypto_alloc_cipher("aes", 0, 0);
```

Здесь *tfm* - это хэндл некоторого преобразования, в данном случае - шифрования блока данных по алгоритму *AES*. Жизненный цикл хэндла *Crypto API* сводится к трём этапам:

- создание/инициализация хэндла
- выполнение необходимых криптоопераций, досупных через хэндл
- уничтожение хэндла

Независимо от используемого алгоритма, через хэндл типа `crypto_cipher` доступны реализации базовых функций *шифрования одного блока* данных, а соответсвующий ему *API* (*Single Block Cipher API*) предоставляет методы установки ключа и зашифрования/расшифрования блока. В качестве самостоятельного *API* он не представляет большого интереса для пользователя, однако, с точки зрения разработчика, он важен, поскольку с его помощью реализована ключевая концепция *Crypto API* - так называемые "шаблоны" (*templates*). Шаблоны используются в связке с базовыми функциями и реализуют:

- режимы блочного шифрования (*ECB*, *CBC*, *CFB*, *CTR*, ...)
- режим ключезависимого хэширования (*hMAC*)
- *AEAD*-режимы шифрования
- функции, не связанные с непосредственной криптообработкой данных, например - распараллеливание (*pcrypt*)

При создании хэндла (см. пример выше), требуемый алгоритм задётся строкой - именем алгоритма, эта строка имеет следующую семантику:

```
template(single block cipher/message digest)
```

Кроме того, там, где это применимо, шаблоны могут быть "завёрнуты" в другие шаблоны:

```
template_a(template_b(single block cipher/message digest))
```

При этом использование шаблонов без базового алгоритма невозможно. Приведу несколько примеров:

- *aes* - базовый алгоритм шифрования блока *AES*
- *cbc(aes)* - алгоритм шифрования *AES* в режиме сцепления блоков шифра (*CBC*)
- *sha1* - алгоритм хэширования *SHA1*
- *hmac(sha1)* - алгоритм ключезависимого хэширования (имитозащиты) с хэш-функцией *SHA1*
- *authenc(hmac(sha1), cbc(aes))* - *AEAD* с алгоритмом шифрования *AES-CBC* и алгоритмом имитозащиты *hMAC(SHA1)*

Применение симметричных блочных шифров в каком-либо режиме осуществляеся через отдельный *API*. Стоит отметить, что сейчас в ядре существуют три таких *API*, однако два из них объявлены устаревшими (*deprecated*), а актуальным является *Symmetric Key Cipher API*. Этот *API* пред оставляет пользователю методы установки ключа/синхропосылки и зашифрования/расшифрования данных произвольной длины, он так же является *неблокирующим* (асинхронным) - методы шифрования максимально быстро возвращают управление вызывающему, в то время как запрос на выполнение криптооперации и функция обратного вызова, используемая для оповещения пользователя о завершении операции, передаются системному планировщику. *Symmetric Key Cipher API* будет подробно рассмотрен, когда мы перейдём к созданию модуля ядра, тестирующего встроенный нами алгоритм.

Помимо двух названных, в *Crypto API* существуют под-*API* для:

- *AEAD*-шифрования (*Authenticated Encryption With Associated Data (AEAD) Cipher API*)
- блокирующего хэширования/имитозащиты (*Synchronous Message Digest API*)
- неблокирующего хэширования/имитозащиты (*Asynchronous Message Digest API*)
- генерации псевдослучайных чисел (*Random Number API*)
- асимметричного шифрования (*Asymmetric Cipher API*)
- согласования ключей (*Key-agreement Protcol Primitives (KPP) Cipher API*)

Для разработчика наличие шаблонов в *Crypto API* означает, что, встраивая новый алгоритм симметричного шифрования, достаточно реализовать лишь базовую функцию шифрования блока, которая, затем, может быть инкапсулирована в соответствующие шаблоны для получения требуемых режимов. **Но это не совсем так.** Действительно, если взглянуть на исходный код включенных в ядро алгоритмов (*crypto/aes_generic.c*, *crypto/blowfish_generic.c*, ...), то можно увидеть, что там реализованы только базовые функции. Однако, если мы таким же образом реализуем функцию шифрования блока классического *ГОСТ 28147-89*, "завернём" её в режим гаммирования (шаблон *CTR*), а после этого полученный алгоритм проверим на тестовых последовательностях, то мы получим неверный результат! Всё дело в том, что режим гаммирования, описанный в *ГОСТ*, отличается от алгоритма гаммирования, реализованного в шаблоне *CTR*. То же самое касается и других национальных алгоритмов, с которыми я имел дело. В таких случаях необходимо встраивать полноценные блочные шифры в нужных режимах, как это сделано, например, в оптимизированной реализации алгоритма *AES* (*AES-NI* - *arch/x86/crypto/aesni-intel_glue.c*). Позже мы рассмотрим оба варианта встраивания.

Пожалуй, это всё, что я хотел сказать по поводу архитектуры. Заинтересованным в более подробном описании следует обратиться к документации, а моего изложения должно быть достаточно, чтобы мы могли двигаться дальше.

[2] Подготовка почвы

Итак, мы готовы к тому, чтобы начать встраивать в *Crpyto API* какой-нибудь новый алгоритм, с той лишь оговоркой, что у нас пока нет алгоритма, который можно было бы встроить. Для этой статьи я не стал утруждать себя реализацией "настоящего" криптоалгоритма, поскольку едва ли смог бы сделать это качественно (оставьте это криптографам). С другой стороны, делать шифр-"пустышку" (*null cipher*) было бы не так интересно, тем более, что такой в ядре уже есть. Таким образом, мы пойдём на компромисс и напишем реализацию элементарного алгоритма шифрования:

$$C_i = P_i \oplus K$$

$$P_i = C_i \oplus K$$

Где:

- C_i — блок шифртекста
- P_i — блок открытого текста
- K — ключ
- $\oplus$ — операция "исключающее ИЛИ" (*XOR*)

Размеры блока и ключа для этого алгоритма примем равными 128 битам (16 байт).

Можем приступать к реализации. Определим криптоконтекст нашего алгоритма и функции создания/уничтожения контекста:

```
#define XOR_CIPHER_KEY_SIZE 16

typedef struct xor_cipher_ctx xor_cipher_ctx;

struct xor_cipher_ctx
{
        uint8_t key[XOR_CIPHER_KEY_SIZE];
};

xor_cipher_ctx* xor_cipher_allocate()
{
        xor_cipher_ctx *cipher = calloc(1, sizeof(xor_cipher_ctx));
        return cipher;
}

void xor_cipher_free(xor_cipher_ctx *ctx)
{
        memset(ctx->key, 0xFF, XOR_CIPHER_KEY_SIZE);
        free(ctx);
}
```

Добавляем методы установки ключа и зашифрования/расшифрования блока:

```
#define XOR_CIPHER_BLOCK_SIZE 16

void xor_cipher_set_key(xor_cipher_ctx *ctx, uint8_t *key)
{
        memmove(ctx->m_key, key, XOR_CIPHER_KEY_SIZE);
}
```

```
void xor_cipher_crypt_block(xor_cipher_ctx *ctx, uint8_t *dst, uint8_t *src)
{
        for (int i = 0; i < XOR_CIPHER_BLOCK_SIZE; i++)
        {
                dst[i] = src[i] ^ ctx->key[i];
        }
}
```

Учитывая обратимость операции "*XOR*", метод `xor_cipher_crypt_block` используется для зашифрования и расшифрования одновременно. Шифрование блока - это хорошо, но станет ещё лучше, если мы реализуем какой-нибудь из режимов блочного шифрования, например, режим сцепления блоков шифра (*CBC*):

$$C_0 = IV$$

$$C_i = E_k(P_i \oplus C_{i-1})$$

$$P_i = C_{i-1} \oplus D_k(C_i)$$

Где:

- IV — синхропосылка
- E_k, D_k — функция зашифрования и расшифрования блока на ключе K соответственно

Продолжим работу, реализуем методы зашифрования и расшифрования в режиме сцепления блоков шифра. Зашифрование:

```
void xor_cipher_encrypt_cbc(xor_cipher_ctx *ctx,
                uint8_t *_dst, uint32_t len, uint8_t *_src, uint8_t *_iv)
{
        uint32_t blocks = len / XOR_CIPHER_BLOCK_SIZE;
        uint32_t leftover = len - (blocks * XOR_CIPHER_BLOCK_SIZE);

        uint8_t *dst = _dst, *src = _src, *iv = _iv;

        for (uint32_t i = 0; i < blocks; i++)
        {
                memmove(dst, src, XOR_CIPHER_BLOCK_SIZE);

                for (int j = 0; j < XOR_CIPHER_BLOCK_SIZE; j++)
                {
                        dst[j] ^= iv[j];
                }

                xor_cipher_crypt_block(ctx, dst, dst);

                iv = dst;
                dst += XOR_CIPHER_BLOCK_SIZE;
                src += XOR_CIPHER_BLOCK_SIZE;
        }

        if (leftover)
        {
                memmove(dst, src, leftover);

                for (uint32_t i = 0; i < leftover; i++)
                {
                        dst[i] ^= iv[i];
                        dst[i] ^= ctx->key[i];
                }
        }
}
```

Расшифрование:

```
void xor_cipher_decrypt_cbc(xor_cipher_ctx *ctx,
                uint8_t *_dst, uint32_t len, uint8_t *_src, uint8_t *_iv)
{
        uint32_t blocks = len / XOR_CIPHER_BLOCK_SIZE;
        uint32_t leftover = len - (blocks * XOR_CIPHER_BLOCK_SIZE);
        uint8_t u[XOR_CIPHER_BLOCK_SIZE], iv[XOR_CIPHER_IV_SIZE];

        uint8_t *dst = _dst, *src = _src;

        memmove(iv, _iv, XOR_CIPHER_IV_SIZE);

        for (uint32_t i = 0; i < blocks; i++)
        {
                memmove(u, src, XOR_CIPHER_BLOCK_SIZE);
                xor_cipher_crypt_block(ctx, dst, src);

                for (int j = 0; j < XOR_CIPHER_BLOCK_SIZE; j++)
                {
                        dst[j] ^= iv[j];
                }

                memmove(iv, u, XOR_CIPHER_IV_SIZE);
                dst += XOR_CIPHER_BLOCK_SIZE;
                src += XOR_CIPHER_BLOCK_SIZE;
        }

        if (leftover)
        {
                for (uint32_t i = 0; i < leftover; i++)
                {
                        dst[i] = src[i] ^ ctx->key[i];
                        dst[i] ^= iv[i];
                }
        }
}
```

Вот это уже интересней! Теперь, пользуясь данной реализацией, мы можем подготовить тестовые последовательности, на которых, в дальнейшем, проверим реализацию этого же алгоритма в ядре. Под спойлером - значения, которые использовал я.

Шифрование блока	
Ключ	`2f 1b 1a c6 d1 be cb a2 f8 45 66 0d d2 97 5c a3`
Тест №1	
Входные данные	`cc 6b 79 0c db 55 4f e5 a0 69 05 96 11 be 8c 15`
Выходные данные	`e3 70 63 ca 0a eb 84 47 58 2c 63 9b c3 29 d0 b6`
Тест №2	
Входные данные	`53 f5 f1 ef 67 a5 ba 6c 68 09 b5 7a 24 de 82 5f`
Выходные данные	`7c ee eb 29 b6 1b 71 ce 90 4c d3 77 f6 49 de fc`

Зашифрование *CBC*	
Ключ	`ec 8d 93 30 69 7e f8 63 0b f5 58 ec de 78 24 f2`
Синхропосылка	`db 02 1f a8 5a 22 15 cf 49 f7 80 8b 7c 24 a1 f3`
Открытый текст	`6e 96 50 42 84 d2 7e e8 44 9b 75 1d e0 ac 0a 58 ee 40 24 cc 32 fc 6e c4 e2 fc d1 f5 76 6a 45 9a e4 88 ba d6 12 07 28 86`
Шифртекст	`59 19 dc da b7 8e 93 44 06 99 ad 7a 42 f0 8f 59 5b d4 6b 26 ec 0c 05 e3 ef 90 24 63 ea e2 ee 31 53 d1 42 c0 97 75 d5 06`

Расшифрование *CBC*	
Ключ	`ec 8d 93 30 69 7e f8 63 0b f5 58 ec de 78 24 f2`
Синхропосылка	`db 02 1f a8 5a 22 15 cf 49 f7 80 8b 7c 24 a1 f3`
Шифртекст	`db e9 1d c6 1f 13 1a 5a 34 2b 90 1e c3 b1 6f e9 52 1b 91 7f 8d 8f 6d b4 42 87 ad 85 5f 2d 89 7d`
Открытый текст	`ec 66 91 5e 2c 4f f7 f6 76 29 48 79 61 ed ea e8 65 7f 1f 89 fb e2 8f 8d 7d 59 65 77 42 e4 c2 66`

Исходники реализации алгоритма и программы тестирования доступны здесь, а мы практически готовы к тому, чтобы перейти в *kernel space*, но перед этим я хочу обратить ваше внимание на один важный момент. Дело в том, что многие известные алгоритмы шифрования не предполагают обработку последнего *неполного* блока входных данных. Так, например, реализации алгоритма *ГОСТ 28147-89 ДОЛЖНЫ* возвращать признак ошибки, если размер входных данных не кратен размеру блока, тогда как белорусский *Belt* предусматривает такую обработку. Мой алгоритм также её предусматривает (ключ и текущее значение синхропосылки усекаются до размеров неполного блока). Этот факт сыграет свою роль немного позже, пока стоит просто иметь это в виду.

[3] Погружение в ядро

Действие переносится в пространство ядра. Программирование в ядре несколько отличается от программирования в пространстве пользователя и, ввиду сложности отладки, является достаточно трудоёмким процессом. Однако, работая над этим материалом, я не ставил перед собой задачу познакомить читателя с основами программирования модулей ядра, ведь об этом и без меня предостаточно написано, в том числе и здесь, на Хабре (ещё; и ещё). Поэтому, если читатель совершенно не знаком с написанием модулей, то я бы рекомендовал сперва просмотреть те материалы, которые я привёл выше, либо поискать самостоятельно (уверяю, это не займёт много времени). А с теми, кто готов идти дальше, мы приступаем к глубокому изучению *Crypto API*.

Итак, у нас есть алгоритм, и мы хотим встроить его в ядро, но с чего же начать? Разумеется, с документации. К сожалению, раздел, посвящённый разработке/встраиванию алгоритмов, даёт лишь очень общие знания об этом процессе, но, по крайней мере, помогает сориентироваться в правильном направлении. Конкретно, мы узнаём о существовании функций, отвечающих за регистрацию и разрегистрацию алгоритмов в ядре. Давайте разбираться:

```
/* include/linux/crypto.h */

int crypto_register_alg(struct crypto_alg *alg);
int crypto_register_algs(struct crypto_alg *algs, int count);

int crypto_unregister_alg(struct crypto_alg *alg);
int crypto_unregister_algs(struct crypto_alg *algs, int count);
```

Эти функции возвращают отрицательное значение в случае ошибки, и 0 - в случае успешного завершения, а (раз)регистрируемый(ые) алгоритм(ы) описываются структурой `crypto_alg`, заглянем в её определение (*include/linux/crypto.h*):

```
/* include/linux/crypto.h */

struct crypto_alg {
        struct list_head cra_list;
        struct list_head cra_users;

        u32 cra_flags;
        unsigned int cra_blocksize;
        unsigned int cra_ctxsize;
        unsigned int cra_alignmask;

        int cra_priority;
        atomic_t cra_refcnt;

        char cra_name[CRYPTO_MAX_ALG_NAME];
        char cra_driver_name[CRYPTO_MAX_ALG_NAME];

        const struct crypto_type *cra_type;

        union {
                struct ablkcipher_alg ablkcipher;
                struct blkcipher_alg blkcipher;
                struct cipher_alg cipher;
                struct compress_alg compress;
        } cra_u;

        int (*cra_init)(struct crypto_tfm *tfm);
        void (*cra_exit)(struct crypto_tfm *tfm);
        void (*cra_destroy)(struct crypto_alg *alg);

        struct module *cra_module;
} CRYPTO_MINALIGN_ATTR;
```

К счастью, эта структура очень хорошо задокументирована, и нам не придётся гадать о значении того или иного поля:

- `cra_flags`: набор флагов, описывающих алгоритм. Флаги определены константами, начинающимися с "`CRYPTO_ALG_`", в *include/linux/crypto.h*, и используются для "тонкой настройки" описания алгоритма
- `cra_blocksize`: байтовый размер блока алгоритма. Все типы преобразований, кроме хэширования, возвращают ошибку при попытке обработать данные, размер которых меньше этого значения
- `cra_ctxsize`: байтовый размер криптоконтекста. Ядро использует это значение при выделении памяти под контекст
- `cra_alignmask`: маска выравнивания для входных и выходных данных. Буферы для входных и выходных данных алгоритма должны быть выровнены по этой маске. Это обязательно для алгоритмов, выполняющихся на "железе", неспособном обращаться к данным по произвольным адресам
- `cra_priority`: приоритет данной реализации алгоритма. Если в ядре зарегистрировано больше одного преобразования с одинаковым `cra_name`, то, при обращении по этому имени, будет возвращён алгоритм с наибольшим приоритетом
- `cra_name`: название алгоритма. Ядро использует это поле для поиска реализаций
- `cra_driver_name`: уникальное имя реализации алгоритма. Если в ядре зарегистрировано больше одного преобразования с одинаковым `cra_name`, и необходимо запросить алгоритм с меньшим приоритетом, то нужно обращаться по этому имени
- `cra_type`: тип криптопреобразования. Указатель на экземпляр структуры типа `crypto_type`, реализующий функции обратного вызова, общие для всех типов преобразований. Доступные варианты: `&crypto_blkcipher_type`, `&crypto_ablkcipher_type`, `&crypto_ahash_type`, `&crypto_rng_type`. Это поле следует оставлять пустым для следующих типов преобразований: шифрования блока (`cipher`), сжатия (`compress`), блокирующего хэширования (`shash`)
- `cra_u`: собственно, реализация алгоритма. Одна из структур союза, должна быть заполнена специфическими функциями, определяемыми типом преобразования. Это поле следует оставлять пустым для блокирующего и неблокирующего хэширования (`ahash`)
- `cra_init`: функция инициализации экземпляра преобразования. Эта функция вызывается единожды, во время создания экземпляра (сразу после выделения памяти под криптоконтекст). В этой функции следует выполнять различные подготовительные действия, если такие необходимы (например, выделение дополнительной памяти, проверка аппаратных возможностей). В ином случае поле можно оставить пустым
- `cra_exit`: деинициализация экземпляра преобразования. В этой функции следует выполнять действия, обратные тем, что были выполнены в `cra_init`, если таковые имели место. В ином случае поле можно оставить пустым
- `cra_module`: владелец этой реализации преобразования. Следует установить значение `THIS_MODULE`

Недокумментированные оставшиеся поля предназначены для внутреннего использования и не должны заполняться. Вроде бы ничего сложного. Для начала, мы хотим встроить реализацию алгоритма шифрования блока, поэтому взглянем ещё на структуру `cipher_alg` из союза `cra_u`:

```
/* include/linux/crypto.h */

struct cipher_alg {
        unsigned int cia_min_keysize;
        unsigned int cia_max_keysize;
        int (*cia_setkey)(struct crypto_tfm *tfm, const u8 *key,
                          unsigned int keylen);
        void (*cia_encrypt)(struct crypto_tfm *tfm, u8 *dst, const u8 *src);
        void (*cia_decrypt)(struct crypto_tfm *tfm, u8 *dst, const u8 *src);
};
```

Тут всё ещё проще и, как мне кажется, не нуждается в разъяснении. Теперь мы готовы к тому, чтобы посмотреть, как всё это работает на практике. Кстати, обратите внимание на то, что сигнатуры функций в структуре `cipher_alg` аналогичны сигнатурам функций из *API* нашего алгоритма из 2 части.

Пишем модуль ядра. Определяем криптоконтекст и функцию установки ключа в соответствии с сигнатурой `cipher_alg.cia_setkey`:

```
#define XOR_CIPHER_KEY_SIZE   16

struct xor_cipher_ctx
{
        u8 key[XOR_CIPHER_KEY_SIZE];
};

static int xor_cipher_setkey(struct crypto_tfm *tfm, const u8 *key,
             unsigned int len)
{
        struct xor_cipher_ctx *ctx = crypto_tfm_ctx(tfm);
        u32 *flags = &tfm->crt_flags;

        if (len != XOR_CIPHER_KEY_SIZE)
        {
                *flags |= CRYPTO_TFM_RES_BAD_KEY_LEN;
                return -EINVAL;
        }

        memmove(ctx->key, key, XOR_CIPHER_KEY_SIZE);
        return 0;
}
```

В *API* нашего алгоритма криптоконтекст передавался функциям напрямую, здесь же функции принимают хэндл алгоритма `tfm`, из которого, затем, контекст извлекается с помощью функции `crypto_tfm_ctx`. Так же, здесь мы проверяем длину переданного ключа. Если длина некорректна, то мы выставляем соответствующий флаг (`CRYPTO_TFM_RES_BAD_KEY_LEN`) и возвращаем код `EINVAL` (22: *Invalid argument*). Теперь, определяем функцию шифрования блока в соответствии с `cipher_alg.cia_encrypt`:

```
static void xor_cipher_crypt(struct crypto_tfm *tfm, u8 *out, const u8 *in)
{
        struct xor_cipher_ctx *ctx = crypto_tfm_ctx(tfm);
        int i;

        for (i = 0; i < XOR_CIPHER_BLOCK_SIZE; i++)
        {
                out[i] = in[i] ^ ctx->key[i];
        }
}
```

Здесь - ничего нового. Так же, как и в оригинальном *API*, мы не будем определять дополнительную функцию для расшифровки, а указатель `cipher_alg.cia_decrypt` просто проинициализируем функцией `xor_cipher_crypt`. Наконец, определяем экземпляр структуры `crypto_alg`, заполняем её и вызываем функции регистрации и разрегистрации алгоритма, соответсвенно, после загрузки и перед выгрузкой модуля:

```
static struct crypto_alg xor_cipher = {
        .cra_name = "xor-cipher",
        .cra_driver_name = "xor-cipher-generic",
        .cra_priority = 100,
        .cra_flags = CRYPTO_ALG_TYPE_CIPHER,
        .cra_blocksize = XOR_CIPHER_BLOCK_SIZE,
        .cra_ctxsize = sizeof(struct xor_cipher_ctx),
        .cra_module = THIS_MODULE,
        .cra_u = {
                .cipher = {
                        .cia_min_keysize = XOR_CIPHER_KEY_SIZE,
                        .cia_max_keysize = XOR_CIPHER_KEY_SIZE,
                        .cia_setkey     = xor_cipher_setkey,
                        .cia_encrypt = xor_cipher_crypt,
                        .cia_decrypt = xor_cipher_crypt
                }
        }
};
```

```
static int __init xor_cipher_init(void)
{
        return crypto_register_alg(&xor_cipher);
}

static void __exit xor_cipher_exit(void)
{
        crypto_unregister_alg(&xor_cipher);
}
```

Учитывая всё вышесказанное, здесь не должно возникать никаких вопросов. Если, скомпилировав и загрузив такой модуль, выполнить в терминале команду "`cat /proc/crypto`", то в выводимом списке зарегистрированных алгоритмов можно отыскать и свой:

```
name         : xor-cipher
driver       : xor-cipher-generic
module       : xor_cipher
priority     : 100
refcnt       : 1
selftest     : passed
internal     : no
type         : cipher
blocksize    : 16
min keysize  : 16
max keysize  : 16
```

Неплохо, а? Но это только начало, и сейчас, я, по идее, должен был перейти к написанию модуля, тестирующего наш алгоритм, но всё же, я решил не смешивать вопросы встраивания и использования. Поэтому тестироваться мы будем позже, а в следующей части мы посмотрим, как можно встроить реализацию нашего алгоритма в режиме сцепления блоков шифра.

[3.1] "Немного более глубокое" погружение в ядро

Забегу немного вперёд. Внимательный читатель помнит о существовании шаблонов. Так вот, реализовав в прошлой части шифрование блока, мы запросто можем обернуть эту реализацию в шаблон *CBC* и получть реализацию нашего алгоритма в режиме сцепления блоков шифра, но, протестировав её на моих последовательностях, вызов шифрования в режиме *CBC* вернёт нам код ошибки 22 (`EINVAL`). И тут самое время вспомнить, что я говорил по поводу обработки неполного блока входных данных и поля `crypto_alg.cra_blocksize` . Дело в том, что реализация режима *CBC* ядра понятия не имеет о том, как обрабатывать неполный блок. Более того, оборачивая наш алгоритм в режим *CBC*, ядро регистрирует новый алгоритм, размер блока которого равен размеру блока базового алгоритма. После вызова "апишной" функции зашифрования алгоритмом *cbc(xor-cipher)*, размер входных данных проверяется на кратность размеру блока и, если они не кратны, функция возвращает `EINVAL` . Размер тестового вектора для шифрования в режиме *CBC* (40 байт) умышленно выбран не кратным размеру блока. По моему мнению, если стандартом шифрования предусматривается обработка неполного блока, а реализация этого не делает, то такая реализация едва ли пройдёт проверку на соответствие стандарту, даже если реализация даёт корректный результат, когда условие кратности выполняется (в данном случае, это так). Поэтому, сейчас мы сделаем полную реализацию режима сцепления блоков шифра для нашего алгоритма. Когда я занимался этим 2 года назад, я встраивал алгоритмы с расчётом на их использование через устаревший ныне *API* блокирующего симметричного шифрования, хотя уже тогда предпочтительным считался неблокирующий *API*, который, на сегодняшний день, так же устарел. Эту реализацию мы будем делать под актуальный *Symmetric Key Cipher API*.

Изучив документацию и реализации других алгоритмов в ядре своей ОС, я обнаружил, что объявление и регистрация таких алгоритмов существенно отличается от того, как это делалось раньше, и осуществляется при помощи части *Crypto API*, не отражённой в документации. Но, имея под рукой образец (*arch/x86/crypto/aesni-intel_glue.c*), разобраться, что к чему, оказалось не так уж и сложно. Итак, у нас здесь несколько иные функции для регистрации/разрегистрации алгоритмов:

```
/* include/crypto/internal/skcipher.h */

int crypto_register_skcipher(struct skcipher_alg *alg);
int crypto_register_skciphers(struct skcipher_alg *algs, int count);

void crypto_unregister_skcipher(struct skcipher_alg *alg);
void crypto_unregister_skciphers(struct skcipher_alg *algs, int count);
```

Заглянем в определение структуры `skcipher_alg` :

```
/* include/crypto/skcipher.h */

struct skcipher_alg {
        int (*setkey)(struct crypto_skcipher *tfm, const u8 *key,
                      unsigned int keylen);
        int (*encrypt)(struct skcipher_request *req);
        int (*decrypt)(struct skcipher_request *req);
        int (*init)(struct crypto_skcipher *tfm);
        void (*exit)(struct crypto_skcipher *tfm);

        unsigned int min_keysize;
        unsigned int max_keysize;
        unsigned int ivsize;
        unsigned int chunksize;
        unsigned int walksize;

        struct crypto_alg base;
};
```

В отличие от документации, в заголовочном файле *include/crypto/skcipher.h* эта структура задокументирована. Так, эта структура содержит экземпляр уже знакомой нам `struct crypto_alg` , описывающей алгоритм, здесь же, функции `init` и `exit` , по своему назначению аналогичны таковым из `crypto_alg` . Новыми, на данный момент, являются поля `chunksize` и `walksize` :

- `chunksize`: по поводу значения этого поля в комментарии к коду сказано, что для всех алгоритмов, за исключением поточных (*stream cipher*), оно должно быть равно размеру блока, а, в случае поточных шифров, оно должно быть равно размеру блока базового (*underlying*) алгоритма. Это, в большей степени, касается блочных алгоритмов в режимах гаммирования. Для таких реализаций фактический размер блока (`skcipher_alg.base.cra_blocksize`) равен единице, а значение поля `chunksize` должно быть равно размеру блока алгоритма, используемого для выработки гаммы
- `walksize`: равен значению `chunksize`, за исключением случаев, когда алгоритм может параллельно обрабатывать несколько блоков, тогда `walksize` может быть больше, чем `chunksize`, но обязательно должен быть кратен ему

Ещё одну неопознанную структуру мы видим в аргументах функций `encrypt` и `decrypt`. Структура `skcipher_request` содержит данные, необходимые для выполнения операции симметричного шифрования: указатели на входные/выходные данные и синхропосылку, хэндл криптопреобразования и так далее. Напрямую обращаться к полям этой структуры нет необходимости, поскольку для работы с ней существует отдельный под-*API*, но, всё же, в ней есть одна особенность, о которой обязательно нужно упомянуть.

На самом деле, эта особенность касается *Crypto API* в целом. Дело в том, что все под-*API* шифрования данных произвольной длины работают со входными/выходными данными не через привычные указатели на байтовые массивы (как было в *API* шифрования блока), а через структуры типа `scatterlist`. Вот, например, функция зашифрования из *Synchronous Block Cipher API*:

```
/* include/linux/crypto.h */

int crypto_blkcipher_encrypt(struct blkcipher_desc *desc,
                struct scatterlist *dst, struct scatterlist *src,
                unsigned int nbytes);
```

Структура `skcipher_request` так же содержит поля `struct scatterlist *src` и `struct scatterlist *dst` для входных и выходных данных соответственно. Посмотрим на основные элементы структуры `scatterlist`:

```
/* include/linux/scatterlist.h */

struct scatterlist {
        /* ... */
        unsigned long   page_link;
        unsigned int    offset;
        unsigned int    length;
        /* ... */
};
```

Экземпляр этой структуры можно проинициализировать указателем на некоторые данные. Например, при помощи вызова функции `sg_init_one`:

```
/* include/linux/scatterlist.h */

void sg_init_one(struct scatterlist *sg, const void *buf, unsigned int buflen);
```

Цитируя отсюда:

> В этой функции определяется страница памяти, с которой "начинается" buf (`page_link`), и определяется смещение указателя buf относительно адреса начала страницы (`offset`).

Таким образом, криптографическая подсистема работает напрямую со страницами памяти. Но это ещё не всё, посмотрите на следующий пример:

```
struct crypto_blkcipher *tfm;
struct blkcipher_dest desc;
struct scatterlist sg[2];
u8 *first_segment, *second_segment;

/* crypto and data allocation... */

sg_init_table(sg, 2);
sg_set_buf(&sg[0], first_segment, len);
sg_set_buf(&sg[1], second_segment, len);

crypto_blkcipher_encrypt(&desc, &sg, &sg, 2*len);
```

Здесь, `first_segment` и `second_segment` указывают на данные, *несмежные* в памяти. Таким образом, *Crypto API* (при помощи специальных вспомогательных структур) в один вызов способно обработать "цепочку" `scatterlist`-ов, содержащую "рассеянные" (*scattered*) данные. Такой подход был изначально заложен в *Crypto API* и обусловлен нуждами подсистемы *IPsec*:

> One of the initial goals of this design was to readily support IPsec, so that processing can be applied to paged skb's without the need for linearization.

На самом деле, `scatterlist` и связанный с ним *API* ядра являются фундаментальной вещью при организиации ввода/вывода с прямым доступом к памяти (*Direct Memory Access*, *DMA I/O*). Они позволяют эффективно осуществлять операции такого ввода/вывода над непрерывными в виртуальной, но "рассеянными" по физической памяти буферами данных, но, впрочем, это уже совсем другая история.

Теперь можем переходить к реализации. Как и в прошлый раз, начинаем с определения функции установки ключа:

```
static int xor_skcipher_setkey(struct crypto_skcipher *tfm, const u8 *key,
                       unsigned int len)
{
        return xor_cipher_setkey(crypto_skcipher_tfm(tfm), key, len);
}
```

Здесь всё просто. Для режима *CBC* мы будем работать с тем же типом криптоконтекста (`struct xor_cipher_ctx`), что и в случае шифрования блока, поэтому мы просто переиспользовали функцию установки ключа, реализованную ранее.

Пишем функции зашифрования и расшифрования:

```
static int cbc_encrypt(struct skcipher_request *req)
{
        struct crypto_tfm *tfm = crypto_skcipher_tfm(crypto_skcipher_reqtfm(req));
        struct xor_cipher_ctx *ctx = crypto_tfm_ctx(tfm);
        struct skcipher_walk walk;
        u32 nbytes;
        int i, blocks;
        u8 *src, *dst, *iv;

        skcipher_walk_virt(&walk, req, true);
        iv = walk.iv;

        while ((nbytes = walk.nbytes) >= XOR_CIPHER_BLOCK_SIZE)
        {
                src = (u8*)walk.src.virt.addr;
                dst = (u8*)walk.dst.virt.addr;
                blocks = nbytes / XOR_CIPHER_BLOCK_SIZE;

                while (blocks)
                {
                        for (i = 0; i < XOR_CIPHER_BLOCK_SIZE; i++)
                        {
                                dst[i] = src[i] ^ iv[i];
                        }
```

```
                        xor_cipher_crypt(tfm, dst, dst);
                        iv = dst;

                        src += XOR_CIPHER_BLOCK_SIZE;
                        dst += XOR_CIPHER_BLOCK_SIZE;
                        blocks--;
                }

                nbytes &= XOR_CIPHER_BLOCK_SIZE - 1;
                skcipher_walk_done(&walk, nbytes);
        }

        if ((nbytes = walk.nbytes))
        {
                src = (u8*)walk.src.virt.addr;
                dst = (u8*)walk.dst.virt.addr;

                for (i = 0; i < nbytes; i++)
                {
                        dst[i] = src[i] ^ iv[i];
                        dst[i] ^= ctx->key[i];
                }

                skcipher_walk_done(&walk, 0);
        }

        return 0;
}
```

```
#define XOR_CIPHER_IV_SIZE 16

static int cbc_decrypt(struct skcipher_request *req)
{
        struct crypto_tfm *tfm = crypto_skcipher_tfm(crypto_skcipher_reqtfm(req));
        struct xor_cipher_ctx *ctx = crypto_tfm_ctx(tfm);
        struct skcipher_walk walk;
        u8 u[XOR_CIPHER_BLOCK_SIZE], iv[XOR_CIPHER_BLOCK_SIZE];
        u32 nbytes;
        int i, blocks;
        u8 *src, *dst;

        skcipher_walk_virt(&walk, req, true);
        memmove(iv, walk.iv, XOR_CIPHER_IV_SIZE);

        while ((nbytes = walk.nbytes) >= XOR_CIPHER_BLOCK_SIZE)
        {
                src = (u8*)walk.src.virt.addr;
                dst = (u8*)walk.dst.virt.addr;
                blocks = nbytes / XOR_CIPHER_BLOCK_SIZE;

                while (blocks)
                {
                        memmove(u, src, XOR_CIPHER_BLOCK_SIZE);
                        xor_cipher_crypt(tfm, dst, src);

                        for (i = 0; i < XOR_CIPHER_BLOCK_SIZE; i++)
                        {
                                dst[i] ^= iv[i];
                        }

                        memmove(iv, u, XOR_CIPHER_IV_SIZE);

                        dst += XOR_CIPHER_BLOCK_SIZE;
                        src += XOR_CIPHER_BLOCK_SIZE;
                        blocks--;
                }
```

```
                nbytes &= XOR_CIPHER_BLOCK_SIZE - 1;
                skcipher_walk_done(&walk, nbytes);
        }

        if ((nbytes = walk.nbytes))
        {
                src = (u8*)walk.src.virt.addr;
                dst = (u8*)walk.dst.virt.addr;

                for (i = 0; i < nbytes; i++)
                {
                        dst[i] = src[i] ^ ctx->key[i];
                        dst[i] ^= iv[i];
                }

                skcipher_walk_done(&walk, 0);
        }

        return 0;
}
```

Здесь, мы, так сказать, "прогуливаемся" по входным данным при помощи вспомогательной структуры `skcipher_walk` и связанных с ней функций. Сперва, экземпляр `skcipher_walk` инициализируется вызовом `skcipher_walk_virt`, а после этого функция `skcipher_walk_done` пересчитвает для нас указатели на очередную непрерывную "порцию" входных/выходных данных (`walk.src.virt.addr` и `walk.dst.virt.addr`) длины `walk.nbytes`, содержащуюся в цепочке `scatterlist`-ов.

Завершаем заполнением экземпляра структуры `skcipher_alg`, и регистрацией/разрегистрацией алгоритма:

```
static struct skcipher_alg cbc_xor_cipher = {
        .base = {
                .cra_name = "cbc(xor-cipher)",
                .cra_driver_name = "cbc-xor-cipher",
                .cra_priority = 400,
                .cra_flags = CRYPTO_ALG_ASYNC,
                .cra_blocksize = 1,
                .cra_ctxsize = sizeof(struct xor_cipher_ctx),
                .cra_module     = THIS_MODULE,
        },
        .min_keysize = XOR_CIPHER_KEY_SIZE,
        .max_keysize = XOR_CIPHER_KEY_SIZE,
        .ivsize = XOR_CIPHER_IV_SIZE,
        .setkey = xor_skcipher_setkey,
        .encrypt = cbc_encrypt,
        .decrypt = cbc_decrypt,
        .chunksize = XOR_CIPHER_BLOCK_SIZE,
};

static int __init xor_cipher_init(void)
{
        crypto_register_alg(&xor_cipher);
        return crypto_register_skcipher(&cbc_xor_cipher);
}

static void __exit xor_cipher_exit(void)
{
        crypto_unregister_alg(&xor_cipher);
        crypto_unregister_skcipher(&cbc_xor_cipher);
}
```

Обратите внимание на значение поля `cra_blocksize`. Это небольшая хитрость, необходимая для того, чтобы вызов `skcipher_walk_done` "отдавал" нам последний неполный блок, если такой есть.

Скомпилировав и загрузив модуль, находим в `/proc/crypto` наш алгоритм:

```
name         : cbc(xor-cipher)
driver       : cbc-xor-cipher
module       : xor_cipher
priority     : 400
refcnt       : 1
selftest     : passed
internal     : no
type         : skcipher
async        : yes
blocksize    : 1
min keysize  : 16
max keysize  : 16
ivsize       : 16
chunksize    : 16
```

Готовый модуль (исходники здесь) регистрирует в ядре два новых алгоритма: "`xor-cipher`" (алгоритм шифрования блока) и "`cbc(xor-cipher)`" (реализация в режиме сцепления блоков шифра). На этом, по встраиванию, у меня всё. Для дальнейшего изучения этого вопроса могу лишь порекомендовать исходный код реализаций, которые уже есть в ядре (*arch/x86/crypto*), а мы переходим к следующей части.

[4] Тестирование

К сожалению, я не смог придумать более остроумного заголовка для этой части, чем просто "Тестирование", да и, в конце концов, это ведь именно то, чем мы будем здесь заниматься. Хотя, непосредственно, тесты - это лишь форма, за которой стоит суть этого раздела: показать, в целом, как *пользоваться Crypto API*. Для этого мы напишем и разберём ещё один модуль и пользовательскую программу, выполняющие серию тестов над нашими, встроенными в ядро, алгоритмами.

Начнём с модуля ядра. Для задания тестовых последовательностей введём структуру `cipher_testvec_t`:

```
typedef enum test_t
{
        TEST_BLK_ENCRYPT = 0,
        TEST_BLK_DECRYPT,
        TEST_CBC_ENCRYPT,
        TEST_CBC_DECRYPT,
        TEST_END,
} test_t;

struct cipher_testvec_t
{
        test_t test;
        u32 len;
        char *key;
        char *iv;
        char *in;
        char *result;
};
```

Значения полей этой структуры тривиальны: входные данные `in`, длиной `len`, подаются на вход алгоритма шифрования блока (`.test = TEST_BLK_*`) либо шифрования в режиме сцепления блоков шифра (`.test = TEST_CBC_*`), с ключом `key` и, где это необходимо, синхропосылкой `iv`. Полученный результатат криптопреобразования сравнивается с эталонным значением `result`. Значение `TEST_END` используется для обозначения окончания массива структур `cipher_testvec_t`.

Ниже - функция тестирования алгоритма шифрования блока:

```
static int test_blk(cipher_testvec_t *testvec)
{
        struct crypto_cipher *tfm = NULL;
        int encrypt = (testvec->test == TEST_BLK_ENCRYPT) ? 1 : 0;
        u8 dst[16];

        tfm = crypto_alloc_cipher("xor-cipher", 0, 0);
        if (IS_ERR(tfm))
        {
                pr_err("error allocating xor-cipher: %ld\n", PTR_ERR(tfm));
                return 0;
        }

        crypto_cipher_setkey(tfm, (u8*)testvec->key, 16);

        if (encrypt)
        {
                crypto_cipher_encrypt_one(tfm, dst, (u8*)testvec->in);
        }
        else
        {
                crypto_cipher_decrypt_one(tfm, dst, (u8*)testvec->in);
        }

        crypto_free_cipher(tfm);
```

```
        if (memcmp(dst, testvec->result, 16))
        {
                pr_err("block %sciphering test failed!\n", encrypt ? "" : "de");
                dumpb((u8*)testvec->key, 16, "key");
                dumpb((u8*)testvec->in, 16, "in");
                dumpb(dst, 16, "result");
                dumpb((u8*)testvec->result, 16, "should be");

                return 0;
        }

        return 1;
}
```

Эта функция крайне проста, в ней используются вполне очевидные вызовы из *Single Block Cipher API*. Сперва, при помощи функции `crypto_alloc_cipher`, мы создаём хэндл криптографического преобразования с именем "*xor-cpher*". После этого, устанавливаем ключ шифрования (`crypto_cipher_setkey`), и, в зависимости от текущего теста, выполняем операцию зашифрования (`crypto_cipher_encrypt_one`) либо расшифрования (`crypto_cipher_decrypt_one`) блока. В завершение, уничтожаем хэндл (`crypto_free_cipher`) и сравниваем результат выполнения криптооперации с эталоном.

С функцией тестирования алгоритма в режиме CBC всё гораздо интереснее:

```
static int test_cbc(cipher_testvec_t *testvec)
{
        struct scatterlist sg;
        struct cb_data_t cb_data;
        struct crypto_skcipher *tfm = NULL;
        struct skcipher_request *req = NULL;
        int encrypt = (testvec->test == TEST_CBC_ENCRYPT) ? 1 : 0;
        u32 err;
        u8 *buf = NULL;

        tfm = crypto_alloc_skcipher("cbc-xor-cipher", 0, 0);
        if (IS_ERR(tfm))
        {
                pr_err("error allocating cbc-xor-cipher: %ld\n", PTR_ERR(tfm));
                goto exit;
        }

        req = skcipher_request_alloc(tfm, GFP_KERNEL);
        if (!req)
        {
                pr_err("error allocating skcipher request\n");
                goto exit;
        }

        buf = kmalloc(testvec->len, GFP_KERNEL);
        if (!buf)
        {
                pr_err("memory allocation error\n");
                goto exit;
        }

        memmove(buf, (u8*)testvec->in, testvec->len);
        sg_init_one(&sg, buf, testvec->len);

        crypto_skcipher_setkey(tfm, (u8*)testvec->key, 16);
        skcipher_request_set_crypt(req, &sg, &sg, testvec->len, (u8*)testvec->iv);

        skcipher_request_set_callback(req, 0, skcipher_cb, &cb_data);
        init_completion(&cb_data.completion);

        err = (encrypt) ? crypto_skcipher_encrypt(req)
                        : crypto_skcipher_decrypt(req);
```

```
        switch (err)
        {
                case 0:
                        break;

                case -EINPROGRESS:
                case -EBUSY:
                        wait_for_completion(&cb_data.completion);
                        err = cb_data.err;
                        if (!err)
                        {
                                break;
                        }

                default:
                        pr_err("failed with error: %d\n", err);
                        goto exit;
        }

        if (memcmp(buf, testvec->result, testvec->len))
        {
                pr_err("cbc %sciphering test failed!\n", encrypt ? "" : "de");
                dumpb((u8*)testvec->key, 16, "key");
                dumpb((u8*)testvec->iv, 16, "iv");
                dumpb((u8*)testvec->in, testvec->len, "in");
                dumpb(buf, testvec->len, "result");
                dumpb((u8*)testvec->result, testvec->len, "should be");

                goto exit;
        }

        skcipher_request_free(req);
        crypto_free_skcipher(tfm);
        kfree(buf);

        return 1;

exit:
        if (buf)
        {
                kfree(buf);
        }
        if (req)
        {
                skcipher_request_free(req);
        }
        if (tfm)
        {
                crypto_free_skcipher(tfm);
        }

        return 0;
}
```

Самое главное: сейчас мы работает с *неблокирующим API* (*Symmetric Key Cipher API*), отсюда несколько иной подход к организации процесса.

Смотрим на функцию `test_cbc`:

- в объявлениях мы видим уже знакомые по прошлым разделам структуры, кроме одной новой: `cb_data_t`, но о ней чуть позже
- выполнение функции начинается с создания ключевых объектов: хэндла алгоритма "*cbc-xor-cipher*" (`crypto_alloc_skcipher`) и экземпляра `skcipher_requset` (`skcipher_request_alloc`)
- затем мы готовим буфер для входных/выходных данных: выделяем участок динамической памяти (`kmalloc`) размером не менее, чем размер входного вектора, и копируем последний в свежевыделенную память; указателем на эту память инициализируем экземпляр `scatterlist` (`sg_init_one`)

- устанавливаем ключ шифрования (`crypto_skcipher_setkey`) и инициализируем экземпляр `skcipher_requset` (`skcipher_request_set_crypt`). Второй и третий параметр функции `skcipher_request_set_crypt` это, соответственно, входные и выходные данные. В качестве аргумента этих параметров мы используем один и тот же экземпляр `scatterlist` , таким образом, шифрование будет выполнено "на месте", а результат криптооперации будет доступен по указателю `buf`
- устанавливаем функцию обратного вызова, для оповещения о завершении криптооперации (`skcipher_request_set_callback`)

Вот с этого момента поподробнее. Поскольку мы имеем дело с неблокирующим *API*, то мы можем оказаться в ситуации, когда вызов функции шифрования вернёт управление раньше, чем данные будут непосредственно обработаны. В этом случае, для оповещения о завершении криптооперации, будет вызана функция обратного вызова, предоставленная пользователем. Эта функция и передаваемые ей произвольные пользовательские данные устанавливаются с помощью вызова `skcipher_request_set_callback` . В нашем примере пользовательские данные имеют тип `cb_data_t` , а функция обратного вызова (`skcipher_cb`) определена следующим образом:

```
struct cb_data_t
{
        struct completion completion;
        int err;
};

static void skcipher_cb(struct crypto_async_request *req, int error)
{
        struct cb_data_t *data = req->data;

        if (error == -EINPROGRESS)
        {
                return;
        }

        data->err = error;
        complete(&data->completion);
}
```

Структура `completion` и связанные с ней функции служат для синхронизации выполнения функции `test_cbc` и, непосредственно шифрования. В конце концов, нам необходимо дождаться окончания обработки, и только после этого возвращать управление из `test_cbc` . Разобравшись с этим, возвращаемся к месту в `test_cbc` , на котором мы остановились:

- инициализируем экземпляр `completion` (`init_completion`), передаваемый функции обратного вызова
- в зависимости от текущего теста, вызываем функцию зашифрования (`crypto_skcipher_encrypt`) или расшифрования (`crypto_skcipher_decrypt`)
- анализируем код возврата функции шифрования:
- - 0 означает, что непосредственно шифрование завершилось успешно
- - коды `-EINPROGRESS` и `-EBUSY` указывают на то, что операция шифрования ещё выполняется. В этом случае, о завершении нас уведомит изменение состояния экземпляра `completion` (`complete` и `wait_for_completion`), переданного функции обратного вызова
- - любой другой код возврата означает ошибку выполения функции `crypto_skcipher_encrypt`

Код после `switch` -а в пояснении не нуждается. На этом всё. Иходный код данного модуля доступен здесь, скомпилировав и загрузив этот модуль (при условии, что загружен модуль с самими алгоритмами), в терминале вы получите что-то вроде:

```
insmod: ERROR: could not insert module xor_cipher_testing.ko: Operation not permitted
```

А в журнале ядра (`dmesg`) появится такая запись:

```
[---] done 4 tests, passed: 4, failed: 0
```

Кстати, в документации есть подобный пример использования *SK Cipher API*, но он недостаточно подробно прокомментирован. Ещё пример - в упомянутой мной в самом начале статье из "Хакера", но там описан *Asynchronous Block Cipher API*, ныне устаревший. И всё-таки, это уже что-то, и заинтересованному читателю есть, где разгуляться.

[4.1] Возвращение в *user-space*

Пришло время посмотреть, как можно воспользоваться криптографией ядра в пользовательском пространстве. Насчёт этого в документации есть отдельный раздел, и, в принципе, его прочтения оказывается достаточно для того, чтобы начать применять эти возможности в своих программах, но это при условии, что вы уже знакомы с интерфейсом сокетов в *Linux*. За примером документация отсылает к библиотеке *libkcapi*, написанной одним из разработчиков *Crypto API* (и соавтором самой документации). Для "общения" с ядром эта библиотека использует *Netlink*-интерфейс, она "заворачивает" низкоуровневую работу с *Netlink* в удобный пользовательский *API*. Если, в своих приложениях, у вас есть необходимость в криптографии ядра, то я бы рекомендовал использовать именно эту библиотеку, тем более, что она до сих пор поддерживается (текущая версия 1.0.3).

Но мы с вами проделали весь этот путь не ради того, чтобы просто воспользоваться библиотекой. Поэтому, дальше мы посмотрим, каким образом код пользовательского пространства, на самом низком уровне (не ниже системных вызовов), взаимодействует с *Crypto API*. Свой пример я сделал на основе плагина *af_alg* из *Strongswan*, его код я также рекомендую к ознакомлению.

Итак, поехали. В целом, в тестовой программе я использовал ту же архитектуру, что и в тестовом модуле. Однако, здесь не будет тестирования алгоритма шифрования блока, поскольку существующий интерфейс позволяет обратиться лишь к полноценным блочным шифрам. Таким образом, у нас остаётся только функция `test_cbc`.

```
static int test_cbc(cipher_testvec_t *testvec)
{
        uint8_t dst[testvec->len];
        int encrypt = (testvec->test == TEST_CBC_ENCRYPT) ? 1 : 0;
        struct af_alg_skcipher *tfm = NULL;

        tfm = af_alg_allocate_skcipher("cbc-xor-cipher");
        if (!tfm)
        {
                fprintf(stderr, "error allocating \"cbc-xor-cipher\"\n");
                goto err;
        }

        if (!af_alg_skcipher_setkey(tfm, (uint8_t*)testvec->key,
                XOR_CIPHER_KEY_SIZE))
        {
                fprintf(stderr, "can't set \"cbc-xor-cipher\" key\n");
                goto err;
        }

        if (!af_alg_skcipher_crypt(tfm, encrypt, dst,
                        testvec->len, (uint8_t*)testvec->in,
                        (uint8_t*)testvec->iv, XOR_CIPHER_IV_SIZE))
        {
                goto err;
        }

        af_alg_free_skcipher(tfm);

        if (memcmp(dst, (uint8_t*)testvec->result, testvec->len))
        {
                fprintf(stderr, "cbc %sciphering test failed!\n",
                        encrypt ? "" : "de");
                dumpb((uint8_t*)testvec->key, XOR_CIPHER_KEY_SIZE, "key");
                dumpb((uint8_t*)testvec->iv, XOR_CIPHER_IV_SIZE, "iv");
                dumpb((uint8_t*)testvec->in, testvec->len, "in");
                dumpb(dst, testvec->len, "result");
                dumpb((uint8_t*)testvec->result, testvec->len, "should be");

                return 0;
        }

        return 1;
```

```
err:
        if (tfm)
        {
                af_alg_free_skcipher(tfm);
        }

        return 0;
}
```

Логика этой функции не должна вызвать вопросов. Мы, как и несколько раз до этого, создаём хэндл, устанавливаем ключ, шифруем данные, уничтожаем хэндл и сравниваем результат с эталоном. Всё интересное здесь скрыто за структурой `af_alg_skcipher` и связанными с ней функциями `af_alg_*`. Смотрим дальше.

```
struct af_alg_skcipher
{
        int sockfd;
};

static struct af_alg_skcipher* af_alg_allocate_skcipher(char *name)
{
        struct af_alg_skcipher *tfm = NULL;
        struct sockaddr_alg sa = {
                .salg_family = AF_ALG,
                .salg_type = "skcipher",
        };

        strncpy((char*)sa.salg_name, name, sizeof(sa.salg_name));

        tfm = calloc(1, sizeof(struct af_alg_skcipher));
        if (!tfm)
        {
                errno = ENOMEM;
                goto err;
        }

        tfm->sockfd = socket(AF_ALG, SOCK_SEQPACKET, 0);
        if (tfm->sockfd == -1)
        {
                goto err;
        }

        if (bind(tfm->sockfd, (struct sockaddr*)&sa, sizeof(sa)) == -1)
        {
                goto err;
        }

        return tfm;

err:
        if (tfm->sockfd > 0)
        {
                close(tfm->sockfd);
        }
        if (tfm)
        {
                free(tfm);
        }

        return NULL;
}
```

Идём по коду:

- заполняем экземпляр структуры `sockaddr_alg` (поясню чуть позже)
- выделяем память под новый хэндл (`calloc`)
- открываем сокет к *Crypto API* (`socket`)

Первый параметр системного вызова `socket` определяет семейство протоколов, которые могут быть использованы для "общения" через открытый сокет. Семейство *AF_ALG* предназначено для взаимодействия с *Crypto API*. Константа *AF_ALG*, обычно, определена в заголовочном файле *sys/socket.h*, если это не так, то её можно определить самостоятельно:

```
#ifndef AF_ALG
#define AF_ALG 38
#endif
```

- устанавливаем алгоритм, с которым мы будем взаимодействовать через открытый сокет (`bind`)

Конкретный алгоритм задаётся заполнением экземпляра структуры `sockaddr_alg` , указатель на этот экземпляр передаётся вторым аргументом системного вызова `bind` :

```
/* sys/socket.h */

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Структура `sockadr` определена следующим образом:

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
}
```

Эта структура нужна лишь для подавления предупреждения компилятора, на самом же деле, её реальный вид определяется семейством протоколов (`sa_family`), для *AF_ALG* это:

```
/* linux/if_alg.h */

struct sockaddr_alg {
        __u16   salg_family;
        __u8    salg_type[14];
        __u32   salg_feat;
        __u32   salg_mask;
        __u8    salg_name[64];
};
```

Мы заполняем три поля:

- `salg_family` : см. выше
- `salg_type` : тип запрашиваемого криптопреобразования: " `skcipher` " - шифрование, " `hash` " - хэширование и имитовставка, " `aead` " - очевидно, *AEAD*, "rng" - генерация псевдослучайных чисел
- `salg_name` : имя алгоритма в ядре (`cra_name` или `cra_driver_name`)

Если запрашиваемый алгоритм не зарегистрирован в ядре, то `bind` вернёт -1, а в переменную `errno` будет помещён код ошибки `ENOENT` .

Таким образом, результатом выполнения функции `af_alg_allocate_skcipher` является открытый сокет, через который можно взаимодействовать с конкретным алгоритмом в ядре. Функция уничтожения хэндла тривиальна и я не буду здесь её приводить.

Теперь посмотрим на функцию установки ключа:

```
static int af_alg_skcipher_setkey(struct af_alg_skcipher *tfm,
                                  uint8_t *key, uint32_t keylen)
{
        return (setsockopt(tfm->sockfd, SOL_ALG, ALG_SET_KEY, key, keylen) == -1)
                        ? 0 : 1;
}
```

Ключ устанавливается при помощи системного вызова `setsockopt`, подробнее о параметрах этого вызова можно узнать на соответствующей *man*-странице. Я лишь отмечу, что константы `SOL_ALG` и `ALG_SET_KEY`, обычно, определены в файлах *sys/socket.h* и *linux/af_alg.h* соответсвенно, если, на вашей системе, это не так, то их можно определить самостоятельно:

```
#ifndef SOL_ALG
#define SOL_ALG 279
#endif

#ifndef ALG_SET_KEY
#define ALG_SET_KEY 1
#endif
```

Наконец, переходим к самому главному:

```
static int af_alg_skcipher_crypt(struct af_alg_skcipher *tfm, int encrypt,
                                 uint8_t *_dst, uint32_t _len, uint8_t *_src,
                                 uint8_t *iv, uint32_t ivlen)
{
        int type = encrypt ? ALG_OP_ENCRYPT : ALG_OP_DECRYPT;
        struct msghdr msg = {};
        struct cmsghdr *cmsg;
        struct af_alg_iv *ivm;
        struct iovec iov;
        char buf[CMSG_SPACE(sizeof(type)) +
                         CMSG_SPACE(offsetof(struct af_alg_iv, iv) + ivlen)];
        int op = 0;
        ssize_t len, remainig = _len;
        uint8_t *src = _src, *dst = _dst;

        op = accept(tfm->sockfd, NULL, 0);
        if (op == -1)
        {
                goto end;
        }

        memset(buf, 0, sizeof(buf));

        /* fill in af_alg cipher controll data */
        msg.msg_control = buf;
        msg.msg_controllen = sizeof(buf);

        /* operation type: encrypt or decrypt */
        cmsg = CMSG_FIRSTHDR(&msg);
        cmsg->cmsg_level = SOL_ALG;
        cmsg->cmsg_type = ALG_SET_OP;
        cmsg->cmsg_len = CMSG_LEN(sizeof(type));
        memmove(CMSG_DATA(cmsg), &type, sizeof(type));

        /* initialization vector */
        cmsg = CMSG_NXTHDR(&msg, cmsg);
        cmsg->cmsg_level = SOL_ALG;
        cmsg->cmsg_type = ALG_SET_IV;
        cmsg->cmsg_len = CMSG_LEN(offsetof(struct af_alg_iv, iv) + ivlen);
        ivm = (void*)CMSG_DATA(cmsg);
        ivm->ivlen = ivlen;
        memmove(ivm->iv, iv, ivlen);

        /* set data stream (scatter/gather list) */
        msg.msg_iov = &iov;
        msg.msg_iovlen = 1;

        while (remainig)
        {
                iov.iov_base = src;
                iov.iov_len = remainig;
```

```
                len = sendmsg(op, &msg, 0);
                if (len == -1)
                {
                        if (errno == EINTR)
                        {
                                continue;
                        }

                        goto end;
                }
                while (read(op, dst, len) != len)
                {
                        if (errno != EINTR)
                        {
                                goto end;
                        }
                }

                src += len;
                remainig -= len;

                /* no iv for subsequent data chunks */
                msg.msg_controllen = 0;
        }

        /* done */
        close(op);
        return 1;

end:
        if (op > 0)
        {
                close(op);
        }

        return 0;
}
```

Функция относительно велика, и я опишу её лишь в общих чертах. Здесь есть много структур, наверняка знакомых тем, кто занимается сетевым кодом, но, даже если это не о вас, общая логика происходящего должна быть понятной. В целом, её можно разделить на 3 блока:

- в первом блоке, с помощью системного вызова `accept`, мы создаём ещё один сокет, через который, в дальнейшем, наш код и ядро будут обмениваться данными
- во втором блоке, мы формируем сообщение (`msghdr`), содержащее некоторую управляющую информацию (`cmsghdr`) и, непосредственно, указатели на входные/выходные данные (`iovec`)

Управляющая информация включает в себя тип выполняемой операции (`ALG_SET_OP`): зашифрование (`ALG_OP_ENCRYPT`) или расшифрование (`ALG_OP_DECRYPT`), и синхропосылку (`ALG_SET_IV` , `af_alg_iv`). Опять же, отмечу, что приведённые константы должны быть определены в файле *linux/if_alg.h*, и, если это не так, то их можно определить следующим образом:

```
#ifndef ALG_SET_IV
#define ALG_SET_IV 2
#endif
#ifndef ALG_SET_OP
#define ALG_SET_OP 3
#endif
#ifndef ALG_OP_DECRYPT
#define ALG_OP_DECRYPT 0
#endif
#ifndef ALG_OP_ENCRYPT
#define ALG_OP_ENCRYPT 1
#endif
```

- наконец, в третьем блоке, используя системные вызовы `sendmsg` и `read`, с сокетом, открытым вызовом `accept`, мы, соответсвенно, отправляем в ядро данные на зашифрование/расшифрование и читаем результат

Вот и всё. Исходники этой тестовой программы доступны здесь. Скомпилировав и запустив её, вы увидите следующее сообщение:

```
done 2 tests, passed: 2, failed: 0
```

Но это - при условии, что загружен наш модуль ядра.

---

[5] Что дальше?

Изначально, я планировал разбить эту статью на две части, но, начав работать, оказалось, что объём материала, который я планировал на вторую часть, значительно меньше того, что я подготовил для первой. И всё же, там были довольно интересные вещи, о которых я хотел бы рассказать. Поэтому после заключения я в виде своеобразных заметок кратко расскажу об этих вещах.

А по основной части - у меня всё. Надеюсь, эта статья оказалась для вас полезным или, по крайней мере, увлекательным чтением. Конечно, это далеко не полное руководство по *Crypto API*. В принципе, по статье можно написать о каждом из существующих под-*API*, при этом в некоторые из них я ещё даже не заглядывал. Но, с другой стороны, этого должно быть достаточно для того, чтобы продолжить изучение самостоятельно, если в этом есть необходимость.

Буду рад, если вы поможете мне сделать этот материал лучше: своё мнение, замечания и предложения оставляйте здесь, в комментариях, отравляйте мне на почту или в личные сообщения. Ещё лучше, если свои исправления вы будете предлагать мне прямо на github. Спасибо за внимание!

Используем шаблоны

Я столько раз говорил о том, что алгоритмы в ядре можно "заворачивать" в шаблоны, но так и не рассказал - *КАК* это сделать. Самое время это исправить! На самом деле всё очень просто. Допустим, в ядре зарегистрирован наш *xor-cipher*, тогда, что бы воспользоваться этим алгоритмом в шаблонном режиме *CBC*, нужно сделать следующий вызов (для *SK Cipher API*, для других - аналогично):

```
struct crypto_skcipher *tfm = crypto_alloc_skcipher("cbc(xor-cipher)", 0, 0);
```

В результате мы получаем хэндл алгоритма *xor-cipher*, завёрнутого в стандартный режим сцепления блоков шифра (*crypto/cbc.c*).

"Но постой! - скажете вы. - А что, если наш модуль регистрирует реализацию с именем *cbc(xor-cipher)*?". "Хороший вопрос", - отвечу я. В этой ситуации ядро вернёт нам алгоритм из нашего модуля, и всё же мы можем получить шаблонную реализацию.

Когда ядро ищет алгоритм по имени, оно поступает следующим образом:

- ищет уже зарегистрированный алгоритм с таким именем
- если такого алгоритма нет, то пытается найти и загрузить модуль ядра с таким именем
- если и это не удалось, то ядро пытается "на лету" сконструировать подходящий алгоритм

На третьем шаге ядро перебирает зарегистрированные шаблоны и алгоритмы. На каждой итерации такого перебора ядро пытается "собрать" новый алгоритм из очередного шаблона и алгоритма. Например, для шаблона "*cbc*", если имена нашего алгоритма:

```
.cra_name = "xor-cipher";
.cra_driver_name = "xor-cipher-generic";
```

То ядро создаст и зарегистрирует алгоритм с именами:

```
.cra_name = "cbc(xor-cipher)";
.cra_driver_name = "cbc(xor-cipher-generic)";
```

В третьей части, когда мы изучали структуру `crypto_alg`, я упоминал о том, что `cra_driver_name` выбирается *уникальным* для каждой реализации, и по нему также можно запросить алгоритм. Для конкретно этого случая это означает, что, если `cra_driver_name` отлично от "*cbc(xor-cipher-generic)*", то шаблонную реализацию можно получить вызовом:

```
struct crypto_skcipher *tfm = crypto_alloc_skcipher("cbc(xor-cipher-generic)", 0, 0);
```

Больше тестов

Кстати, мои тестовые программы и модули выглядят так, как они выглядят, не просто потому что мне так захотелось. Разумеется, в *Crypto API* есть модуль, отвечающий за тестирование, и свои я сделал по его образу и подобию.

Этот модуль называется *tcrypt* (*crypto/tcrypt.c*) и с ним есть одна проблема. Дело в том, что точка входа и тестирование скорости находятся прямо в *crypto/tcrypt.c*, а вот реализация тестов известного ответа расположена отдельно, в файле *crypto/testmgr.c*. В процессе сборки ядра, *testmgr* чаще всего компилируется статично в ядро и лишь экспортирует в глобальное пространство имён функцию, отвечающую за тестирование (а на моей системе тестирование вовсе отключено в конфигурации ядра).

Из сказанного следует, что просто так добавить свои тесты в *tcrypt* не получится. Поэтому я немного "подшаманил" исходники *tcrypt* и *testmgr*, вынес их из дерева ядра и написал *Makefile*, компилирующий их в один модуль - *tcryptext (tcrypt External)*. Это добро можно забрать у меня на гите.

Если вы хотите протестировать свой алгоритм в ядре через *tcryptext* (тоже самое работает и с *tcrypt*, только с пересборкой ядра), то просто добавьте соответствующий код в исходники модуля по аналогии с каким-либо алгоритмом того же типа, что и ваш. После этого скомпилируйте и загрузите модуль с нужными параметрами (о параметрах в *README.md*, в репозитории).

Файл *testmgr.c* также рекомендую как пример использования всех видов преобразований в *Crypto API*.

Домик на дереве

Допустим, вы разрабочик *Linux-based* операционной системы и вы поддерживаете собственную кастомную ветку ядра. Тогда свою криптографию (если она у вас есть) вы бы также могли поддерживать внутри дерева каталогов исходных кодов ядра.

И с этим нет никаких проблем. Просто разместите файлы с исходным кодом вашего криптографического модуля внутри папки "*crypto*" и добавьте соответствующие записи в файлы *crypto/Kconfig* и *Makefile*. Например, если в *crypto* мы поместили файл *xor_cipher.c*, то в файл *crypto/Kconfig* (под строкой *comment "Ciphers"*) добавим нечто вроде:

```
...
config CRYPTO_XOR_CIPHER
        tristate "XOR cipher algorithm"
        help
          Custom XOR cipher algorithm.
...
```

А в файл *crypto/Makefile*:

```
...
obj-$(CONFIG_CRYPTO_XOR_CIPHER) += xor_cipher.o
...
```

После этого в меню кофнигурации ядра (раздел "*Cryptographic API*") появится пункт для выбора нашего алгоритма. Это также позволит выбрать для него вариант статичной компиляции в образ ядра, что разумно в ряде случаев.

И не забудьте позаботиться о добавлении соответствующих тестов в *tcrypt* и *testmgr*.

Полезные ссылки

Литература:

- Authenticated encryption
- Linux Kernel Crypto API
- Crypto API в ядре Linux
- Режим сцепления блоков шифротекста
- Пишем простой модуль ядра Linux
- Работаем с модулями ядра в Linux
- Учимся писать модуль ядра (Netfilter) или Прозрачный прокси для HTTPS
- Scatterlist Cryptographic API
- How SKBs work
- The chained scatterlist API
- libkcapi - Linux Kernel Crypto API User Space Interface Library
- setsockopt(2) - Linux man page

Git:

- Эта статья и примеры
- tcryptext
- Strongswan af_alg plugin
- Linux kernel