内核加密机制

linux 3.3.8

总体框架

仅显示关键结构

内核加密机制在内核中保存两个全局链表crypto\_template\_list和crypto\_alg\_list

crypto\_template\_list 保存所有的加密方式

e.g cbc ecb

crypto\_alg\_list 保存所有的加密算法

e.g md5 sha128 aes

crypto\_template\_list由各个算法通过crypto\_register\_template函数添加到链表中

crypto\_alg\_list由各个算法通过crypto\_register\_alg函数添加到链表中

数据结构说明

crypto\_alg结构可以理解为真正保存加密接口的一个通用结构

struct list\_head cra\_users;

指向crypto\_spawn结构所在的链表

union cra\_u;

union {

struct ablkcipher\_alg ablkcipher;

struct aead\_alg aead;

struct blkcipher\_alg blkcipher;

struct cipher\_alg cipher;

struct compress\_alg compress;

struct rng\_alg rng;

} cra\_u;

xxxxxx\_alg结构体说明

保存实际算法实现的函数接口和算法相关信息

blkcipher\_alg

struct blkcipher\_alg {

int (\* setkey) (struct crypto\_tfm \*tfm, const u8 \*key,unsigned int keylen);

int (\* encrypt) (struct blkcipher\_desc \*desc,struct scatterlist \*dst, struct scatterlist \*src,unsigned int nbytes);

int (\* decrypt) (struct blkcipher\_desc \*desc,struct scatterlist \*dst, struct scatterlist \*src,unsigned int nbytes);

const char \* geniv;

unsigned int min\_keysize;

unsigned int max\_keysize;

unsigned int ivsize;

};

int (\* setkey)

将产生的密钥及其相关信息保存在crypto\_tfm结构的\_\_crt\_ctx[]部分。

同它也负责检查密钥长度的合法性，以防在cra\_init函数调用时，使用密钥长度与实际设置长度不符。

int (\* encrypt)

加密存放在scatterlist里面的数据，需要对齐scatterlist并且确保每一个加密块大小已经设置好

这个函数不应该去修改\_\_crt\_ctx[]部分数据

int (\* decrypt)

环境与encrypt一致 过程与encrypt相反

union中结构初始化流程

cipher

首先由算法本身实现xxxxx\_alg结构体的初始化，通过xxx\_mod\_init函数调用crypto\_register\_xxxxx函数将xxxxx\_alg传递到crypto\_alg结构体中

通过\_\_crypto\_register\_alg函数将当前crypto\_alg结构体添加到crypto\_alg\_list链表当中

至此加密算法在内核中已成功注册

ablkcipher/rng/aead/blkcipher/compress

首先由算法本身实现crypto\_alg结构体的初始化，使用crypto\_register\_alg函数调用\_\_crypto\_register\_alg函数

将当前crypto\_alg结构体添加到crypto\_alg\_list链表当中，至此xxxxx\_alg算法在内核中已成功注册

crypto\_template结构是通用组合算法的模板

struct hlist\_head instances;

该字段保存temp下所有的实例

struct crypto\_instance \*(\*alloc)(struct rtattr \*\*tb);

每个算法在该函数中保存自己的模板

crypto\_instance结构就是加密方式的一个实例对象 通过函数XXXX\_alloc\_instance

struct crypto\_alg alg;

该算法实现的函数接口

struct crypto\_template \*tmpl;

算法实例对应的模板

void \*\_\_ctx[];

保存crypto\_spawn结构

crypto\_spawn结构保存加密算法和加密方式的所有信息，以便之后通过crypto\_spawn\_tfm生成crypto\_tfm的对象

struct crypto\_alg \*alg;

保存的加密算法的接口

struct crypto\_instance \*inst;

保存的加密方式的接口

Ps.以上结构体在内存中一直存在

crypto\_tfm结构是内核中真正组合算法的实例化对象，真正的操作对象

union crt\_u;

union {

struct ablkcipher\_tfm ablkcipher;

struct aead\_tfm aead;

struct blkcipher\_tfm blkcipher;

struct cipher\_tfm cipher;

struct hash\_tfm hash;

struct compress\_tfm compress;

struct rng\_tfm rng;

} crt\_u;

保存算法能够正常使用的最基本参数和函数接口

void \*\_\_crt\_ctx[];

保存加密算法注册的结构体的相关内容

常用来保存加密算法密钥的相关信息

算法组合流程

仅涉及密钥加解密过程时，不需要算法的组合

crypto\_xxxx\_alloc

| run kthread(cryptomgr\_probe)

+---->crypto\_get\_attr\_alg-------------------------------> cryptomgr\_probe[get crypto\_alg]

|

+---->crypto\_alloc\_instance----> crypto\_init\_spawn[combine crypto\_instance and crypto\_alg]

函数中数据结构的传递

e.g xxxx = shash

xxxxxx\_desc

|

+--->crypto\_xxxxx

|

+---->crypto\_tfm

|

+---> xxxx\_tfm

|

+---> crypto\_alg

|

+----> xxxx\_alg

Ecryptfs中的使用

加密算法

确定算法

使用crypto\_alloc\_hash函数设置使用的算法

crypto\_alloc\_hash(ECRYPTFS\_DEFAULT\_HASH, 0, CRYPTO\_ALG\_ASYNC);

/\* ECRYPTFS\_DEFAULT\_HASH 该字段设置算法 \*/

模块注册

init update final详细流程

以MD5为例介绍内核加密模块在内核中的注册过程

/\* crypto/md5.c \*/

static struct shash\_alg alg = {

.digestsize = MD5\_DIGEST\_SIZE,

.init = md5\_init,

.update = md5\_update,

.final = md5\_final,

.export = md5\_export,

.import = md5\_import,

.descsize = sizeof(struct md5\_state),

.statesize = sizeof(struct md5\_state),

.base = {

.cra\_name = "md5",

.cra\_flags = CRYPTO\_ALG\_TYPE\_SHASH,

.cra\_blocksize = MD5\_HMAC\_BLOCK\_SIZE,

.cra\_module = THIS\_MODULE,

}

};

首先由算法本身实现shash\_alg结构体的初始化，通过md5\_mod\_init函数调用crypto\_register\_shash函数将shash\_alg传递到crypto\_alg结构体中

通过\_\_crypto\_register\_alg函数将当前crypto\_alg结构体添加到crypto\_alg\_list链表当中

至此加密算法在内核中已成功注册

相关结构体

函数调用关系

crypto\_alloc\_hash函数将crypto\_alg与shash\_alg连接起来

static struct crypto\_alg \*\_\_crypto\_alg\_lookup(const char \*name, u32 type,

u32 mask)

{

struct crypto\_alg \*q, \*alg = NULL;

int best = -2;

/\* 遍历crypto\_alg\_list链表 查找符合条件的q并返回 \*/

list\_for\_each\_entry(q, &crypto\_alg\_list, cra\_list) {

int exact, fuzzy;

if (crypto\_is\_moribund(q))

continue;

if ((q->cra\_flags ^ type) & mask)

continue;

if (crypto\_is\_larval(q) &&

!crypto\_is\_test\_larval((struct crypto\_larval \*)q) &&

((struct crypto\_larval \*)q)->mask != mask)

continue;

exact = !strcmp(q->cra\_driver\_name, name);

fuzzy = !strcmp(q->cra\_name, name);

if (!exact && !(fuzzy && q->cra\_priority > best))

continue;

if (unlikely(!crypto\_mod\_get(q)))

continue;

best = q->cra\_priority;

if (alg)

crypto\_mod\_put(alg);

alg = q;

if (exact)

break;

}

return alg;

}

static int crypto\_init\_ops(struct crypto\_tfm \*tfm, u32 type, u32 mask)

{

const struct crypto\_type \*type\_obj = tfm->\_\_crt\_alg->cra\_type;

if (type\_obj)

return type\_obj->init(tfm, type, mask);/\* 此步骤调用shash\_compat\_init函数 \*/

switch (crypto\_tfm\_alg\_type(tfm)) {

case CRYPTO\_ALG\_TYPE\_CIPHER:

return crypto\_init\_cipher\_ops(tfm);

case CRYPTO\_ALG\_TYPE\_COMPRESS:

return crypto\_init\_compress\_ops(tfm);

default:

break;

}

BUG();

return -EINVAL;

}

结构体传递

至此找到MD5所对应的shash\_alg结构体

加密方式

确定算法

通过ecryptfs\_crypto\_api\_algify\_cipher\_name函数设置加密方式

ecryptfs\_crypto\_api\_algify\_cipher\_name(&full\_alg\_name, crypt\_stat->cipher, "cbc");

/\* crypt\_stat->cipher为加密算法 cbc为加密模式 \*/

模块注册

以cbc(aes)为例介绍在内核模块中的注册流程

static struct crypto\_template crypto\_cbc\_tmpl = {

.name = "cbc",

.alloc = crypto\_cbc\_alloc,

.free = crypto\_cbc\_free,

.module = THIS\_MODULE,

};

首先在结构体中初始化相关字段保存在crypto\_template结构体中，通过crypto\_cbc\_module\_init函数调用crypto\_register\_template函数将crypto\_template结构体插入crypto\_template\_list链表中以完成注册

相关结构体

函数调用关系

将crypto\_alg与blkcipher\_alg对应起来

ecb

在ecryptfs\_process\_key\_cipher函数中初始化为ecb模式

---------------------

原文：https://blog.csdn.net/code\_\_L/article/details/79718318