Linux加密框架设计与实现(转)

一、 前言

Linux加密框架是内核安全子系统的重要组成部份，同时，它又一个的独立子系统形式出现，从它出现在内核根目录下的crypto/就可以看出其地位了。
Crypto实现较为复杂，其主要体现在其OOP的设计思路和高度的对像抽像与封装模型，作者展现了其出色的架构设计水准和面向对像的抽像能力。本文力图从加密框架的重要应用，即IPSec(xfrm)的两个重要协议AH和ESP对加密框架的使用，展现其设计与实现。
内核版本：2.6.31.13

二、 算法模版

1. 模版的基本概念

算法模版是加密框架的第一个重要概念。内核中有很多算法是动态生成的，例如cbc(des)算法。内核并不存在这样的算法，它事实上是cbc和des的组合，但是内核加密框架从统一抽像管理的角度。将cbc(des)看做一个算法，在实际使用时动态分配并向内核注册该算法。这样，可以将cbc抽像为一个模版，它可以同任意的加密算法进行组合。算法模版使用结构crypto_template来描述，其结构原型：

点击(此处)折叠或打开

```
1. struct crypto_template {
2.         struct list_head list; //模版链表成员，用于注册
3.         struct hlist_head instances; //算法实例链表首部
4.         struct module *module; //模块指针
5.
6.         struct crypto_instance *(*alloc)(struct rtattr **tb); //算法实例分配
7.         void (*free)(struct crypto_instance *inst); //算法实例释放
8.
9.         char name[CRYPTO_MAX_ALG_NAME]; //模版名称
10. };
```

例如，一个名为cbc的算法模版，可以用它来动态分配cbc(des),cbc(twofish)……诸如此类。

crypto/algapi.c下包含了模版的一些常用操作。最为常见的就是模版的注册与注销，其实质是对以crypto_template_list为首的链表的操作过程：

点击(此处)折叠或打开

```
1. static LIST_HEAD(crypto_template_list);
2.
3. int crypto_register_template(struct crypto_template *tmpl)
4. {
```

随笔档案 (201)


```
5.      struct crypto_template *q;
6.      int err = -EEXIST;
7.
8.      down_write(&crypto_alg_sem);
9.
10.     //遍历crypto_template_list，看当前模板是否被注册
11.     list_for_each_entry(q, &crypto_template_list, list) {
12.             if (q == tmpl)
13.                     goto out;
14.     }
15.
16.     //注册之
17.     list_add(&tmpl->list, &crypto_template_list);
18.     //事件通告
19.     crypto_notify(CRYPTO_MSG_TMPL_REGISTER, tmpl);
20.     err = 0;
21. out:
22.     up_write(&crypto_alg_sem);
23.     return err;
24. }
25. EXPORT_SYMBOL_GPL(crypto_register_template);
```

注销算法模版，除了模版本身，还有一个重要的内容是处理算法模版产生的算法实例，关于算法实例，后文详述。

点击(此处)折叠或打开

```
1. void crypto_unregister_template(struct
   crypto_template *tmpl)
2. {
3.      struct crypto_instance *inst;
4.      struct hlist_node *p, *n;
5.      struct hlist_head *list;
6.      LIST_HEAD(users);
7.
8.      down_write(&crypto_alg_sem);
9.
10.     BUG_ON(list_empty(&tmpl->list));
11.     //注销算法模版，并重新初始化模版的list成员
12.     list_del_init(&tmpl->list);
13.
14.     //首先移除模版上的所有算法实例
15.     list = &tmpl->instances;
16.     hlist_for_each_entry(inst, p, list, list) {
17.             int err = crypto_remove_alg(&inst->alg, &users);
18.             BUG_ON(err);
19.     }
20.
21.     crypto_notify(CRYPTO_MSG_TMPL_UNREGISTER, tmpl);
22.
23.     up_write(&crypto_alg_sem);
24.
25.     //释放模版的所有算法实例分配的内存
26.     hlist_for_each_entry_safe(inst, p, n, list, list) {
27.             BUG_ON(atomic_read(&inst->alg.cra_refcnt) != 1);
```

2010年12月(5)
2010年11月(2)
2010年10月(7)
2010年8月(3)

文章档案 (13)

2012年7月(1)
2012年6月(1)
2011年9月(10)
2011年4月(1)

相册 (0)

hoy

最新评论

1. Re:Linux信号（signal) 机制分析
mark
--godfather007
2. Re:Linux文件系统性能优化(转)
服了，调优真是拥有Linux大师级的功底
--Jame-Mei
3. Re:Syscall系统调用Linux内核跟踪
@ rzhftglibc库中对相应系统调用函数的.c文件中都有一行，weak_alias（_xxx,xxx）,是对函数弱符号的一个define，其实都是一个东西。...
--athreee
4. Re:PF_NETLINK应用实例NETLINK_KOBJECT_UEVENT具体实现－－udev实现原理
@ find_answer因为buf是不连续的字符串（中间含有'\0'），所以要进行以下处理其中size为recv函数的返回值static void print_char_buffer(char * ...
--oo153
5. Re:Linux内存管理(上)
讲的太好了
--0dtc0

阅读排行榜

1. Linux信号（signal) 机制分析(98806)
2. Android JNI知识简介(80564)
3. Linux驱动中，probe函数何时被调用(36498)
4. strtok()和strtok_r()(29118)
5. Linux内存管理(上)(25345)

评论排行榜

1. Android JNI知识简介(13)
2. PF_NETLINK应用实例NETLINK_KOBJECT_UEVENT具体实现－－udev实现原理(3)
3. Linux内存管理(上)(3)

```
28.             tmpl->free(inst);
29.         }
30.         crypto_remove_final(&users);
31. }
32. EXPORT_SYMBOL_GPL(crypto_unregister_template);
```

2. 算法模版的查找

点击(此处)折叠或打开

```
1. crypto_lookup_template函数根据名称，查找相应的模版:
2.
3. struct
   crypto_template *crypto_lookup_template(const char *name)
4. {
5.         return
   try_then_request_module(__crypto_lookup_template(name),
   name);
6. }
```

__crypto_lookup_template完成实质的模版模找工作，而try_then_request_module则尝试动态插入相应的内核模块，如果需要的话:

点击(此处)折叠或打开

```
1. static struct
   crypto_template *__crypto_lookup_template(const char *nam
   e)
2. {
3.         struct crypto_template *q, *tmpl = NULL;
4.
5.         down_read(&crypto_alg_sem);
6.         //遍历crypto_template_list链，匹备模版名称
7.         list_for_each_entry(q, &crypto_template_list, list) {
8.                 if (strcmp(q->name, name))
9.                         continue;
10.                //查找命中，需要对其增加引用，以防止其正在使用时，模
   块被卸载。完成该操作后返回查找到的模版
11.                if (unlikely(!crypto_tmpl_get(q)))
12.                        continue;
13.
14.                tmpl = q;
15.                break;
16.        }
17.        up_read(&crypto_alg_sem);
18.
19.        return tmpl;
20. }
```

3. 模版的算法实例分配时机

模版可以看做一个静态的概念，其只有被动态创建后才具有生命力，本文将模版通过alloc分配创建的算法（对像）称为"实例(instance)"。

算法模版的核心作用是，上层调用者构造一个完整合法的算法名称，如hmac(md5)，触发模版的alloc动作，为该名称分配一个算法实例，类似于为

4. Linux信号（signal) 机制分析(3)
5. strtok()和strtok_r()(2)

推荐排行榜
1. Linux信号（signal) 机制分析(20)
2. Android JNI知识简介(14)
3. Linux驱动中，probe函数何时被调用(5)
4. Linux设备驱动开发环境的搭建（转）(4)
5. Linux内存管理(上)(4)

类实例化一个对像，最终的目的还是使用算法本身。对于xfrm来说，一个典型的算法模版的实例分配触发流程如下所述：
xfrm包裹了一层加密框架支持，参后文“ xfrm加密框架”一节，其算法查找函数为xfrm_find_algo，它调用crypto_has_alg函数进行算法的查找，以验证自己支持的算法是否被内核支持，如xfrm支持cbc(des)，但此时并不知道内核是否有这个算法(如果该算法首次被使用，则还没有分配算法实例)。
crypto_has_alg会调用crypto_alg_mod_lookup完成查找工作，crypto_alg_mod_lookup函数查找不命中，会调用crypto_probing_notify函数进行请求探测：

点击(此处)折叠或打开

```
1. struct
   crypto_alg *crypto_alg_mod_lookup(const char *name, u32
   type, u32 mask)
2. {
3.         ……
4.         ok = crypto_probing_notify(CRYPTO_MSG_ALG_REQUES
   T, larval);
5.         ……
6. }
```

请求是通过通知链表来通告的：

点击(此处)折叠或打开

```
1. int crypto_probing_notify(unsigned long val, void *v)
2. {
3.         int ok;
4.
5.         ok = blocking_notifier_call_chain(&crypto_chain, val, v);
6.         if (ok == NOTIFY_DONE) {
7.                 request_module("cryptomgr");
8.                 ok = blocking_notifier_call_chain(&crypto_chain, va
   l, v);
9.         }
10.
11.        return ok;
12. }
```

在algboss.c中注册了一个名为cryptomgr_notifier的通告块结构，其通告处理函数为cryptomgr_notify

点击(此处)折叠或打开

```
1. static struct notifier_block cryptomgr_notifier = {
2.         .notifier_call = cryptomgr_notify,
3. };
4.
5. static int __init cryptomgr_init(void)
6. {
7.         return crypto_register_notifier(&cryptomgr_notifier);
8. }
9.
10. static void __exit cryptomgr_exit(void)
```

```
{
        int err = crypto_unregister_notifier(&cryptomgr_notifier)
;
        BUG_ON(err);
}
```

这样，当有算法被使用的时候，会调用通告块的处理函数cryptomgr_notify，因为此时的消息是CRYPTO_MSG_ALG_REQUEST，所以cryptomgr_schedule_probe进行算法的探测：

点击(此处)折叠或打开

```
static int cryptomgr_notify(struct
notifier_block *this, unsigned long msg,
                        void *data)
{
        switch (msg) {
        case CRYPTO_MSG_ALG_REQUEST:
                return cryptomgr_schedule_probe(data);
……

        return NOTIFY_DONE;
}
```

cryptomgr_schedule_probe启动一个名为cryptomgr_probe的内核线程来进行算法模版的探测：

点击(此处)折叠或打开

```
static int cryptomgr_schedule_probe(struct
crypto_larval *larval)
{
        ……
        //构造param，以供后面使用
        ……
        thread = kthread_run(cryptomgr_probe, param, "crypto
mgr_probe");
        ……
}
```

cryptomgr_probe完成具体的算法探测过程：

点击(此处)折叠或打开

```
static int cryptomgr_probe(void *data)
{
        struct cryptomgr_param *param = data;
        struct crypto_template *tmpl;
        struct crypto_instance *inst;
        int err;

        //查找算法模版
        tmpl = crypto_lookup_template(param->template);
        if (!tmpl)
                goto err;
```

```
12.
13.         //循环调用模版的alloc函数分配算法实列，并将模版注册之
14.         //这里值得注意的是循环的条件，当返回码为-EAGAIN时，会循环再次尝试
15.         //这样使用的一个场景后面会分析到
16.         do {
17.                 inst = tmpl->alloc(param->tb);
18.                 if (IS_ERR(inst))
19.                         err = PTR_ERR(inst);
20.                 else if ((err = crypto_register_instance(tmpl, inst))
    )
21.                         tmpl->free(inst);
22.         } while (err == -EAGAIN && !signal_pending(current));
23.
24.         //查找中会增加引用，这里已经用完了释放之
25.         crypto_tmpl_put(tmpl);
26.
27.         if (err)
28.                 goto err;
29.
30. out:
31.         kfree(param);
32.         module_put_and_exit(0);
33.
34. err:
35.         crypto_larval_error(param->larval, param->otype, param->omask);
36.         goto out;
37. }
```

理解了算法的注册与查找后，再来理解这个函数就非常容易了，其核心在do{}while循环中，包含了算法实例的分配和注册动作。针对每一种算法模版，其alloc动作不尽一致。后文会对xfrm使用的算法模版一一阐述。

为什么不把“算法实例”直接称之为“算法”，这是因为实例包含了更多的内容，其由结构struct crypto_instance可以看出：

点击(此处)折叠或打开

```
1. struct crypto_instance {
2.         struct crypto_alg alg; //对应的算法名称
3.
4.         struct crypto_template *tmpl; //所属的算法模版
5.         struct hlist_node list; //链表成员
6.
7.         void *__ctx[] CRYPTO_MINALIGN_ATTR; //上下文信息指针
8. };
```

内核使用struct crypto_alg描述一个算法（该结构在后文使用时再来分析），可见一个算法实例除了包含其对应的算法，还包含更多的内容。

当分配成功后，cryptomgr_probe会调用crypto_register_instance将其注册，以期将来可以顺利地找到并使用它：

点击(此处)折叠或打开

```
int crypto_register_instance(struct crypto_template *tmpl,
                             struct crypto_instance *inst)
{
        struct crypto_larval *larval;
        int err;

        //对算法进行合法性检查，并构造完整的驱动名称
        err = crypto_check_alg(&inst->alg);
        if (err)
                goto err;

        //设置算法内核模块指针指向所属模版
        inst->alg.cra_module = tmpl->module;

        down_write(&crypto_alg_sem);

        //注册算法实例对应的算法
        larval = __crypto_register_alg(&inst->alg);
        if (IS_ERR(larval))
                goto unlock;

        //成功后，将算法再注册到所属的模版上面
        hlist_add_head(&inst->list, &tmpl->instances);
        //设置模版指针
        inst->tmpl = tmpl;

unlock:
        up_write(&crypto_alg_sem);

        err = PTR_ERR(larval);
        if (IS_ERR(larval))
                goto err;

        crypto_wait_for_test(larval);
        err = 0;

err:
        return err;
}
```

注册的一个重要工作，就是调用__crypto_register_alg将实例所对应的算法注册到加密框架子系统中。算法注册成功后，上层调用者就可以调用crypto_alg_mod_lookup等函数进行查找，并使用该算法了。

三、 HMAC

MAC（消息认证码）与hash函数非常相似，只是生成固定长度的消息摘要时需要秘密的密钥而已。
HAMC是密钥相关的哈希运算消息认证码（keyed-Hash Message Authentication Code）,HMAC运算利用哈希算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出。具体的算法描述详见：
http://baike.baidu.com/view/1136366.htm?fr=ala0_1。
根据HMAC的特点(可以和类似md5、sha等hash算法组合，构造出

hmac(md5)这样的算法)，Linux 加密框架将其抽像为一个算法模版。本章将假设上层调用者使用了名为hmac(md5)的算法，展示这一算法是如何被构造、初始化及调用以实现数据验证的。

1. 算法模版的注册与注销

点击(此处)折叠或打开

```
1. static struct crypto_template hmac_tmpl = {
2.         .name = "hmac",
3.         .alloc = hmac_alloc,
4.         .free = hmac_free,
5.         .module = THIS_MODULE,
6. };
```

点击(此处)折叠或打开

```
1. static int __init hmac_module_init(void)
2. {
3.         return crypto_register_template(&hmac_tmpl);
4. }
```

点击(此处)折叠或打开

```
1. static void __exit hmac_module_exit(void)
2. {
3.         crypto_unregister_template(&hmac_tmpl);
4. }
```

模版的注册与注销前文已经描述过了。

2. 算法实例的分配

当一个算法需要被使用却查找不到的时候，会尝试调用其模版对应分配相应的算法实列，这也适用于hmac，其alloc函数指针指向hmac_alloc：

点击(此处)折叠或打开

```
1. static struct crypto_instance * hmac_alloc (struct rtattr **tb)
2. {
3.         struct crypto_instance *inst;
4.         struct crypto_alg *alg;
5.         int err;
6.         int ds;
7.
8.         //类型检查，所属算法必需为hash类型
9.         err = crypto_check_attr_type(tb, CRYPTO_ALG_TYPE_HA
   SH);
10.        if (err)
11.                return ERR_PTR(err);
12.
13.        //根据参数名称，查找相应的子算法，如md5,shax等
14.        alg = crypto_get_attr_alg(tb, CRYPTO_ALG_TYPE_HASH,
15.                                  CRYPTO_ALG_TYPE_HASH_MASK);
```

```
16.         //查找失败
17.         if (IS_ERR(alg))
18.                 return ERR_CAST(alg);
19.
20.         //初始化算法实例
21.         inst = ERR_PTR(-EINVAL);
22.
23.         //计算算法实列的消息摘要大小(输出大小)
24.         ds = alg->cra_type == &crypto_hash_type ?
25.             alg->cra_hash.digestsize :
26.             alg->cra_type ?
27.             __crypto_shash_alg(alg)->digestsize :
28.             alg->cra_digest.dia_digestsize;
29.         if (ds > alg->cra_blocksize)
30.                 goto out_put_alg;
31.
32.         //分配一个算法实列，这样，一个新的算法，如hmac(md5)就横空出世了
33.         inst = crypto_alloc_instance("hmac", alg);
34.         //分配失败
35.         if (IS_ERR(inst))
36.                 goto out_put_alg;
37.
38.         //初始化算法实例，其相应的成员等于其子算法中的对应成员
39.         //类型
40.         inst->alg.cra_flags = CRYPTO_ALG_TYPE_HASH;
41.         //优先级
42.         inst->alg.cra_priority = alg->cra_priority;
43.         //计算消息摘要的块长度(输入大小)
44.         inst->alg.cra_blocksize = alg->cra_blocksize;
45.         //对齐掩码
46.         inst->alg.cra_alignmask = alg->cra_alignmask;
47.         //类型指针指向crypto_hash_type
48.         inst->alg.cra_type = &crypto_hash_type;
49.         //消息摘要大小
50.         inst->alg.cra_hash.digestsize = ds;
51.
52.         //计算算法所需的上下文空间大小
53.         inst->alg.cra_ctxsize = sizeof(struct hmac_ctx) +
54.                                 ALIGN(inst->alg.cra_blocksize * 2 + ds,
55.                                       sizeof(void *));
56.
57.         //初始化和退出函数
58.         inst->alg.cra_init = hmac_init_tfm;
59.         inst->alg.cra_exit = hmac_exit_tfm;
60.
61.         //置相应hash算法的操作函数，包含hash函数标准的init/update/final和digest/setkey
62.         inst->alg.cra_hash.init = hmac_init;
63.         inst->alg.cra_hash.update = hmac_update;
64.         inst->alg.cra_hash.final = hmac_final;
65.         //消息摘要函数
66.         inst->alg.cra_hash.digest = hmac_digest;
67.         //setkey(密钥设置函数)
68.         inst->alg.cra_hash.setkey = hmac_setkey;
```

```
69.
70. out_put_alg:
71.         crypto_mod_put(alg);
72.         return inst;
73. }
```

每个模版的alloc动作虽不同，但是它们基本上遵循一些共性的操作：
1、 合法性检验，如类型检查；
2、 取得其子算法（即被模版所包裹的算法，如hmac(md5)中，就是md5）的算法指针；
3、 调用crypto_alloc_instance分配一个相应的算法实列；
4、 对分配成功的算法实例进行实始化，这也是理解该算法实例最核心的部份，因为它初始化算法运行所需的一些必要参数和虚函数指针；

crypto_alloc_instance(algapi.c) 函数用于分配一个算法实例，这个函数有两个重要功能，一个是分配内存空间，另一个是初始化spawn。

点击(此处)折叠或打开

```
1. //name: 模版名称
2. //alg:模版的子算法
3. struct
   crypto_instance *crypto_alloc_instance(const char *name,
4.                                        struct crypto_alg *alg)
5. {
6.         struct crypto_instance *inst;
7.         struct crypto_spawn *spawn;
8.         int err;
9.
10.        //分配一个算法实例，crypto_instance结构的最后一个成员ctx是一个指针变量，所以，在分配空间的时候，在其尾部追加相应的空间，可以使用ctx访问之。
11.        //另一个重要的概念是，算法实例中包含了算法，这个分配，同时也完成了算法实例对应的算法的分配工作。
12.        inst = kzalloc(sizeof(*inst) + sizeof(*spawn), GFP_KERNEL);
13.        if (!inst)
14.                return ERR_PTR(-ENOMEM);
15.
16.        err = -ENAMETOOLONG;
17. //构造完成的算法名称
18.        if (snprintf(inst->alg.cra_name, CRYPTO_MAX_ALG_NAME, "%s(%s)", name,
19.                     alg->cra_name) >= CRYPTO_MAX_ALG_NAME)
20.                goto err_free_inst;
21.
22.        //构造完整的算法驱动名称
23.        if (snprintf(inst->alg.cra_driver_name, CRYPTO_MAX_ALG_NAME, "%s(%s)",
24.                     name, alg->cra_driver_name) >= CRYPTO_MAX_ALG_NAME)
25.                goto err_free_inst;
26.
27.        //spawn指向算法实例的上下文成员，可以这样做是因为__ctx是一个可变长的成员，在分配实例的时候，
```

```
28.         //在尾部增加了一个spawn的空间
29.         spawn = crypto_instance_ctx(inst);
30.         //初始化spawn
31.         err = crypto_init_spawn(spawn, alg, inst,
32.                                 CRYPTO_ALG_TYPE_MASK | CRYPTO_AL
    G_ASYNC);
33.
34.         if (err)
35.                 goto err_free_inst;
36.
37.         return inst;
38.
39. err_free_inst:
40.         kfree(inst);
41.         return ERR_PTR(err);
42. }
```

crypto_instance_ctx取出算法实例的ctx指针，返回值是void *，这意味着可以根具不同的需要，将其转换为所需的类型：

点击(此处)折叠或打开

```
1. static inline void *crypto_instance_ctx(struct
   crypto_instance *inst)
2. {
3.         return inst->__ctx;
4. }
```

一个算法实例被分配成员后，其会被注册至加密子系统，这样，一个算法，例如，hmac(md5)就可以直接被使用了。

3. 待孵化的卵

已经看到了从模版到算法实例的第一层抽像，每个算法在每一次被使用时，它们的运行环境不尽相同，例如，可能会拥有不同的密钥。将算法看成一个类，则在每一次运行调用时，需要为它产生一个"对像"，这在内核中被称为transform，简称为tfm。后文会详细看到分配一个tfm的过程，现在引入这一概念，主要是为了分析spawn。
加密或认证算法，在调用时，都需要分配其算法对应的tfm，在分配算法实例的同时，并没有为之分配相应的tfm结构，这是因为真正的算法还没有被调用，这并不是进行tfm结构分配的最佳地点。在初始化算法实例的时候，加密框架使用了XXX_spawn_XXX函数簇来解决这一问题。这样的算法对像，被称为spawn(卵)。也就是说，在算法实例分配的时候，只是下了一个蛋（设置好spawn），等到合适的时候来对其进行孵化，这个"合适的时候"，通常指为调用算法实际使用的时候。

在crypto_alloc_instance分配算法实例的时候，就顺便分配了spawn，然后调用crypto_init_spawn对其进行初始化：

点击(此处)折叠或打开

```
1. int crypto_init_spawn(struct crypto_spawn *spawn, struct
   crypto_alg *alg,
2.                       struct crypto_instance *inst, u32 mask)
3. {
4.         int err = -EAGAIN;
```

```
5.
6.      //初始化其成员
7.      spawn->inst = inst;
8.      spawn->mask = mask;
9.
10.     down_write(&crypto_alg_sem);
11.     if (!crypto_is_moribund(alg)) {
12.             //加入链表，每个spawn，都被加入到算法的cra_users
    链，即算做算法的一个用户
13.             list_add(&spawn->list, &alg->cra_users);
14.             //spawn的alg成员指针指向当前成员，这就方便引用了
15.             spawn->alg = alg;
16.             err = 0;
17.     }
18.     up_write(&crypto_alg_sem);
19.
20.     return err;
21. }
```

所以，所谓算法的spawn的初始化，就是初始化crypto_spawn结构，核心的操作是设置其对应的算法实例、算法，以及一个加入算法的链表的过程。

4.　　算法的初始化

有了算法实例，仅表示内核拥有这一种“算法”——加引号的意思是说，它可能并不以类似md5.c这样的源代码形式存现，而是通过模版动态创建的。实际要使用该算法，需要为算法分配“运行的对像”，即tfm。

4.1　　tfm

内核加密框架中，使用结构crypto_alg来描述一个算法，每一个算法(实例)相当于一个类，在实际的使用环境中，需要为它分配一个对像，在内核加密框架中，这个“对像”被称为transform（简称tfm）。transform意味“变换”，可能译为“蜕变”更为合适。作者对它的注释是：

```
/*
* Transforms: user-instantiated objects which encapsulate
algorithms
* and core processing logic.  Managed via crypto_alloc_*() and
* crypto_free_*(), as well as the various helpers below.
……
*/
```

tfm是加密框架中一个极为重要的概念，它由结构crypto_tfm描述：

点击(此处)折叠或打开

```
1. struct crypto_tfm {
2.
3.      u32 crt_flags;
4.
5.      union {
6.              struct ablkcipher_tfm ablkcipher;
7.              struct aead_tfm aead;
8.              struct blkcipher_tfm blkcipher;
9.              struct cipher_tfm cipher;
10.             struct hash_tfm hash;
11.             struct ahash_tfm ahash;
```

```
        struct compress_tfm compress;
        struct rng_tfm rng;
    } crt_u;

    void (*exit)(struct crypto_tfm *tfm);

    struct crypto_alg *__crt_alg;

    void *__crt_ctx[] CRYPTO_MINALIGN_ATTR;
};
```

这些成员的作用，将在后面一一看到，值得注意的是，针对每种算法不同，结构定义了一个名为crt_u的联合体，以对应每种算法的tfm的具体操作，例如加密/解密，求hash，压缩/解压等，加密框架引入了一组名为xxx_tfm的结构封装，xxx表示算法类型，也就是crt_u成员。其定义如下：

点击(此处)折叠或打开

```
struct ablkcipher_tfm {
    int (*setkey)(struct
crypto_ablkcipher *tfm, const u8 *key,
                  unsigned int keylen);
    int (*encrypt)(struct ablkcipher_request *req);
    int (*decrypt)(struct ablkcipher_request *req);
    int (*givencrypt)(struct skcipher_givcrypt_request *req);
    int (*givdecrypt)(struct skcipher_givcrypt_request *req);

    struct crypto_ablkcipher *base;

    unsigned int ivsize;
    unsigned int reqsize;
};

struct aead_tfm {
    int (*setkey)(struct crypto_aead *tfm, const u8 *key,
                  unsigned int keylen);
    int (*encrypt)(struct aead_request *req);
    int (*decrypt)(struct aead_request *req);
    int (*givencrypt)(struct aead_givcrypt_request *req);
    int (*givdecrypt)(struct aead_givcrypt_request *req);

    struct crypto_aead *base;

    unsigned int ivsize;
    unsigned int authsize;
    unsigned int reqsize;
};

struct blkcipher_tfm {
    void *iv;
    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);
```

```
36.         int (*decrypt)(struct blkcipher_desc *desc, struct
    scatterlist *dst,
37.                        struct scatterlist *src, unsigned int nbytes);
38. };
39.
40. struct cipher_tfm {
41.         int (*cit_setkey)(struct crypto_tfm *tfm,
42.                           const u8 *key, unsigned int keylen);
43.         void (*cit_encrypt_one)(struct
    crypto_tfm *tfm, u8 *dst, const u8 *src);
44.         void (*cit_decrypt_one)(struct
    crypto_tfm *tfm, u8 *dst, const u8 *src);
45. };
46.
47. struct hash_tfm {
48.         int (*init)(struct hash_desc *desc);
49.         int (*update)(struct hash_desc *desc,
50.                       struct scatterlist *sg, unsigned int nsg);
51.         int (*final)(struct hash_desc *desc, u8 *out);
52.         int (*digest)(struct hash_desc *desc, struct
    scatterlist *sg,
53.                       unsigned int nsg, u8 *out);
54.         int (*setkey)(struct crypto_hash *tfm, const u8 *key,
55.                       unsigned int keylen);
56.         unsigned int digestsize;
57. };
58.
59. struct ahash_tfm {
60.         int (*init)(struct ahash_request *req);
61.         int (*update)(struct ahash_request *req);
62.         int (*final)(struct ahash_request *req);
63.         int (*digest)(struct ahash_request *req);
64.         int (*setkey)(struct crypto_ahash *tfm, const u8 *key,
65.                       unsigned int keylen);
66.
67.         unsigned int digestsize;
68.         unsigned int reqsize;
69. };
70.
71. struct compress_tfm {
72.         int (*cot_compress)(struct crypto_tfm *tfm,
73.                             const u8 *src, unsigned int slen,
74.                             u8 *dst, unsigned int *dlen);
75.         int (*cot_decompress)(struct crypto_tfm *tfm,
76.                               const u8 *src, unsigned int slen,
77.                               u8 *dst, unsigned int *dlen);
78. };
79.
80. struct rng_tfm {
81.         int (*rng_gen_random)(struct
    crypto_rng *tfm, u8 *rdata,
82.                               unsigned int dlen);
83.         int (*rng_reset)(struct
    crypto_rng *tfm, u8 *seed, unsigned int slen);
84. };
```

为了直接访问这些成员，定义了如下宏：

点击(此处)折叠或打开

```
#define crt_ablkcipher crt_u.ablkcipher
#define crt_aead crt_u.aead
#define crt_blkcipher crt_u.blkcipher
#define crt_cipher crt_u.cipher
#define crt_hash crt_u.hash
#define crt_ahash crt_u.ahash
#define crt_compress crt_u.compress
#define crt_rng crt_u.rng
```

这样，要访问hash算法的hash成员，就可以直接使用crt_hash，而不是crt_u.hash。

每种算法访问tfm都使用了二次封装，例如：

点击(此处)折叠或打开

```
struct crypto_ablkcipher {
        struct crypto_tfm base;
};

struct crypto_aead {
        struct crypto_tfm base;
};

struct crypto_blkcipher {
        struct crypto_tfm base;
};

struct crypto_cipher {
        struct crypto_tfm base;
};

struct crypto_comp {
        struct crypto_tfm base;
};

struct crypto_hash {
        struct crypto_tfm base;
};

struct crypto_rng {
        struct crypto_tfm base;
};
```

其base成员就是相应算法的tfm。因为它们拥有相应的起始地址，可以很方便地强制类型转换来操作，内核为此专门定义了一组函数，以hash为例，完成这一工作的是crypto_hash_cast：

点击(此处)折叠或打开

```
1. static inline struct crypto_hash *__crypto_hash_cast(struct
   crypto_tfm *tfm)
2. {
3.         return (struct crypto_hash *)tfm;
4. }
5.
6. static inline struct crypto_hash *crypto_hash_cast(struct
   crypto_tfm *tfm)
7. {
8.         BUG_ON((crypto_tfm_alg_type(tfm) ^
   CRYPTO_ALG_TYPE_HASH) &
9.                 CRYPTO_ALG_TYPE_HASH_MASK);
10.        return __crypto_hash_cast(tfm);
11. }
```

当然，针对各种不同的算法，还有许多不同的XXX_cast函数。这些cast函数，将tfm强制转换为其所属的算法类型的封装结构。

4.2 tfm的分配
对于算法的实始化，其核心功能就是分配一个tfm，并设置其上下文环境，例如密钥等参数，然后初始化上述struct xxx_tfm结构。对于hash类的算法来讲，分配tfm是由crypto_alloc_hash(crypt.h) 这个API来完成的，以AH为例，在其初始化过程中有：

点击(此处)折叠或打开

```
1. static int ah_init_state(struct xfrm_state *x)
2. {
3.         struct crypto_hash *tfm;
4. ……
5. tfm = crypto_alloc_hash(x->aalg-
   >alg_name, 0, CRYPTO_ALG_ASYNC);
6.         if (IS_ERR(tfm))
7.                 goto error;
8. ……
9. }
```

AH调用crypto_alloc_hash为SA中指定的算法（如hmac(md5)）分配一个tfm，第二个参数为0，第三个参数指明了AH使用异步模式。

点击(此处)折叠或打开

```
1. static inline struct
   crypto_hash *crypto_alloc_hash(const char *alg_name,
2.                                u32 type, u32 mask)
3. {
4.         //初始化相应的类型的掩码
5.         type &= ~CRYPTO_ALG_TYPE_MASK; //清除类型的
   CRYPTO_ALG_TYPE_MASK位
6.         mask &= ~CRYPTO_ALG_TYPE_MASK; //清除掩码的
   CRYPTO_ALG_TYPE_MASK位
7.         type |= CRYPTO_ALG_TYPE_HASH; //置类型
   CRYPTO_ALG_TYPE_HASH位
8.         mask |= CRYPTO_ALG_TYPE_HASH_MASK; //置掩码
   CRYPTO_ALG_TYPE_HASH_MASK位
```

```
9.
10.         //最终的分配函数是crypto_alloc_base，它分配一个base(每个算法的tfm)，再将其强制类型转换为所需要结构类型
11.         return __crypto_hash_cast(crypto_alloc_base(alg_name, type, mask));
12. }
```

crypto_alloc_base首先检查相应的算法是否存在，对于hmac(md5)这个例子，xfrm在SA的增加中，会触发相应的算法查找，最终会调用hmac模版的alloc分配算法实例（当然也包括算法本身），然后向内核注册算法及算法实例，所以，查找会命中。接下来的工作，是调用tfm的核心分配函数__crypto_alloc_tfm进行分配，其实现如下：

点击(此处)折叠或打开

```
1. struct crypto_tfm *crypto_alloc_base(const char *alg_name, u32 type, u32 mask)
2. {
3.         struct crypto_tfm *tfm;
4.         int err;
5.
6.         for (;;) {
7.                 struct crypto_alg *alg;
8.
9.                 //根据算法名称，查找相应的算法,它会首先尝试已经加载的算法，如果失败，也会尝试
10.                 //动态插入内核模块
11.                 alg = crypto_alg_mod_lookup(alg_name, type, mask);
12.                 //查找失败，返回退出循环
13.                 if (IS_ERR(alg)) {
14.                         err = PTR_ERR(alg);
15.                         goto err;
16.                 }
17.
18.                 //查找成功，为算法分配tfm
19.                 tfm = __crypto_alloc_tfm(alg, type, mask);
20.                 //分配成功，返回之
21.                 if (!IS_ERR(tfm))
22.                         return tfm;
23.
24.                 //释放引用计算，因为查找会增加引用
25.                 crypto_mod_put(alg);
26.                 //获取返回错误值，根据其值，决定是否要继续尝试
27.                 err = PTR_ERR(tfm);
28.
29. err:
30.                 if (err != -EAGAIN)
31.                         break;
32.                 if (signal_pending(current)) {
33.                         err = -EINTR;
34.                         break;
35.                 }
```

```
        }

        return ERR_PTR(err);
}
```

__crypto_alloc_tfm是内核加密框架中又一重要的函数，它完成了对算法tfm的分配和初始化的工作：

点击(此处)折叠或打开

```
struct crypto_tfm *__crypto_alloc_tfm(struct
crypto_alg *alg, u32 type,
                                      u32 mask)
{
        struct crypto_tfm *tfm = NULL;
        unsigned int tfm_size;
        int err = -ENOMEM;

        //计算tfm所需的空间大小，它包括了tfm结构本身和算法上下文大小
        tfm_size = sizeof(*tfm) + crypto_ctxsize(alg, type, mask);
        //分配tfm
        tfm = kzalloc(tfm_size, GFP_KERNEL);
        if (tfm == NULL)
                goto out_err;

        //__crt_alg成员指向其所属的算法，对于hmac而言，它就是hmac(xxx)，例如hmac(md5)
        tfm->__crt_alg = alg;

        //初始化tfm选项
        err = crypto_init_ops(tfm, type, mask);
        if (err)
                goto out_free_tfm;

        //调用算法的初始化函数，初始化tfm，这有个先决条件是tfm本身没有exit函数的实现
        if (!tfm->exit && alg->cra_init && (err = alg->cra_init(tfm)))
                goto cra_init_failed;

        goto out;

cra_init_failed:
        crypto_exit_ops(tfm);
out_free_tfm:
        if (err == -EAGAIN)
                crypto_shoot_alg(alg);
        kfree(tfm);
out_err:
        tfm = ERR_PTR(err);
out:
        return tfm;
}
```

crypto_init_ops负责初始化tfm的选项，对于一个真正的算法（例如md5、dst）和一个伪算法（我说的“伪”，是指由模版动态分配的，如hmac(xxx)，authenc(xxx,xxx)），因为并不存在这样的算法，只是内核的一个抽像，故称为"伪"，它们的初始化过程是截然不同的。一个伪算法，它都设置了其所属的类型cra_type，例如，对于hmac(xxx)而言，它指向了crypto_hash_type。这样，初始化时，实质上调用的是其所属类型的init函数：

点击(此处)折叠或打开

```
1. static int crypto_init_ops(struct crypto_tfm *tfm, u32
   type, u32 mask)
2. {
3.       //获取tfm所属算法的所属类型
4.       const struct crypto_type *type_obj = tfm->__crt_alg-
   >cra_type;
5.
6.       //如果设置了类型，调用类型的init
7.       if (type_obj)
8.             return type_obj->init(tfm, type, mask);
9.
10.      //否则，判断算法的类型，调用相应的初始化函数，这些在不同的
   算法实现中分析
11.      switch (crypto_tfm_alg_type(tfm)) {
12.      case CRYPTO_ALG_TYPE_CIPHER:
13.            return crypto_init_cipher_ops(tfm);
14.
15.      case CRYPTO_ALG_TYPE_DIGEST:
16.            if ((mask & CRYPTO_ALG_TYPE_HASH_MASK) !=
17.                CRYPTO_ALG_TYPE_HASH_MASK)
18.                  return crypto_init_digest_ops_async(tfm);
19.            else
20.                  return crypto_init_digest_ops(tfm);
21.
22.      case CRYPTO_ALG_TYPE_COMPRESS:
23.            return crypto_init_compress_ops(tfm);
24.
25.      default:
26.            break;
27.      }
28.
29.      BUG();
30.      return -EINVAL;
31. }
```

算法类型的概念很好理解，因为若干个hmac(xxx)都拥有一此相同的类型属性（其它伪算法同样如此），所以可以将它们抽像管理。
对于hash类型的算法而言，它们拥有一个共同的类型crypto_hash_type，其定义在hash.c中：

点击(此处)折叠或打开

```
1. const struct crypto_type crypto_hash_type = {
2.       .ctxsize = crypto_hash_ctxsize,
3.       .init = crypto_init_hash_ops,
```

```
#ifdef CONFIG_PROC_FS
        .show = crypto_hash_show,
#endif
};
```

它的init函数指针指向crypto_init_hash_ops：

点击(此处)折叠或打开

```
static int crypto_init_hash_ops(struct crypto_tfm *tfm, u32 type, u32 mask)
{
        struct hash_alg *alg = &tfm->__crt_alg->cra_hash;

        //其消息摘要大小不同超过1/8个页面
        if (alg->digestsize > PAGE_SIZE / 8)
                return -EINVAL;

        //根据掩码位，判断是同步初始化还是异步，对于crypto_alloc_hash调用下来的而言，它
        //设置了CRYPTO_ALG_TYPE_HASH_MASK位，所以是同步初始化
        if ((mask & CRYPTO_ALG_TYPE_HASH_MASK) != CRYPTO_ALG_TYPE_HASH_MASK)
                return crypto_init_hash_ops_async (tfm);
        else
                return crypto_init_hash_ops_sync(tfm);
}
```

在我们AH的例子中，AH使用了异步模式，所以crypto_init_hash_ops_async会被调用。

前述hash_tfm结构封装了hash类型的算法的通用的操作：

点击(此处)折叠或打开

```
struct hash_tfm {
        int (*init)(struct hash_desc *desc);
        int (*update)(struct hash_desc *desc,
                      struct scatterlist *sg, unsigned int nsg);
        int (*final)(struct hash_desc *desc, u8 *out);
        int (*digest)(struct hash_desc *desc, struct scatterlist *sg,
                      unsigned int nsg, u8 *out);
        int (*setkey)(struct crypto_hash *tfm, const u8 *key,
                      unsigned int keylen);
        unsigned int digestsize;
};
```

先来看同步模式的初始化操作，crypto_init_hash_ops_sync函数负责初始化这一结构：

点击(此处)折叠或打开

```
static int crypto_init_hash_ops_sync(struct crypto_tfm *tfm)
```

```
2. {
3.         struct hash_tfm *crt = &tfm->crt_hash;
4.         struct hash_alg *alg = &tfm->__crt_alg->cra_hash;
5.
6.         //置tfm相应操作为算法本身的对应操作，
7.         //对于hmac(xxx)算法而言，这些东东在hmac_alloc中已经初始
   化过了，也就是hmac_init等函数
8.         crt->init = alg->init;
9.         crt->update = alg->update;
10.        crt->final = alg->final;
11.        crt->digest = alg->digest;
12.        crt->setkey = hash_setkey;
13.        crt->digestsize = alg->digestsize;
14.
15.        return 0;
16. }
```

异步模式则稍有不同，它使用了hash类型算法的通用函数：

点击(此处)折叠或打开

```
1. static int crypto_init_hash_ops_async(struct crypto_tfm *tfm)
2. {
3.         struct ahash_tfm *crt = &tfm->crt_ahash;
4.         struct hash_alg *alg = &tfm->__crt_alg->cra_hash;
5.
6.         crt->init = hash_async_init;
7.         crt->update = hash_async_update;
8.         crt->final = hash_async_final;
9.         crt->digest = hash_async_digest;
10.        crt->setkey = hash_async_setkey;
11.        crt->digestsize = alg->digestsize;
12.
13.        return 0;
14. }
```

不论是同步还是异步，算法的tfm都得到的相应的初始化。回到__crypto_alloc_tfm中来，__crypto_alloc_tfm函数的最后一步是调用算法的cra_init函数(如果它存在的话)，对于hmac(xxx)而言，它在分配的时候指向hmac_init_tfm。hmac_init_tfm的主要工作就是对hmac(xxx)的spawn进行孵化操作。还记得“待孵化的卵”吗？前面讲了只是初始化它，现在到了孵化的时候了

点击(此处)折叠或打开

```
1. static int hmac_init_tfm(struct crypto_tfm *tfm)
2. {
3.         struct crypto_hash *hash;
4.         //因为算法实例的第一个成员就是alg，在注册算法时，就是注册
   的它，所以可以很方便地通过tfm的__crt_alg强制类型转换得到对应的
   算法实例
5.         struct crypto_instance *inst = (void *)tfm->__crt_alg;
6.         //取得算法实例的__ctx域，也就是spawn
7.         struct
   crypto_spawn *spawn = crypto_instance_ctx(inst);
```

```
        //取得tfm的上下文指针
        struct
    hmac_ctx *ctx = hmac_ctx(__crypto_hash_cast(tfm));

        //对hmac(xxx)进行孵化，以hmac(md5)为例，这将得到一个
        md5算法的tfm，当然，通过强制类型转换，它被封装在结构
        crypto_hash中
        hash = crypto_spawn_hash(spawn);
        if (IS_ERR(hash))
                return PTR_ERR(hash);

        //设置子算法指向孵化的tfm
        ctx->child = hash;
        return 0;
}
```

crypto_spawn_hash展示了如何对hash算法簇进行spawn的孵化操作：

点击(此处)折叠或打开

```
static inline struct crypto_hash *crypto_spawn_hash(struct
crypto_spawn *spawn)
{
        //初始化孵化所需的类型和掩码
        u32 type = CRYPTO_ALG_TYPE_HASH;
        u32 mask = CRYPTO_ALG_TYPE_HASH_MASK;

        //调用crypto_spawn_tfm孵化一个tfm，并强制类型转换
        return
__crypto_hash_cast(crypto_spawn_tfm(spawn, type, mask));
}
```

最后的任务交给了crypto_spawn_tfm函数，它为算法孵化一个tfm，因为spawn的alg成员指向了所要孵化的算法，使得这一操作很容易实现

点击(此处)折叠或打开

```
struct crypto_tfm *crypto_spawn_tfm(struct
crypto_spawn *spawn, u32 type,
                                    u32 mask)
{
        struct crypto_alg *alg;
        struct crypto_alg *alg2;
        struct crypto_tfm *tfm;

        down_read(&crypto_alg_sem);
        //要孵化的spawn所属的算法
        alg = spawn->alg;
        alg2 = alg;
        //查找算法所属模块
        if (alg2)
                alg2 = crypto_mod_get(alg2);
        up_read(&crypto_alg_sem);

        //如果其所属模块没了，则标注算法为DYING，出错退回
```

```
18.         if (!alg2) {
19.                 if (alg)
20.                         crypto_shoot_alg(alg);
21.                 return ERR_PTR(-EAGAIN);
22.         }
23.
24.         //初始化tfm
25.         tfm = ERR_PTR(-EINVAL);
26.         //验证掩码标志位
27.         if (unlikely((alg->cra_flags ^ type) & mask))
28.                 goto out_put_alg;
29.
30.         //为算法分配相应的tfm，这样，一个算法的spawn就孵化完成了
31.         tfm = __crypto_alloc_tfm(alg, type, mask);
32.         if (IS_ERR(tfm))
33.                 goto out_put_alg;
34.
35.         return tfm;
36.
37. out_put_alg:
38.         crypto_mod_put(alg);
39.         return tfm;
40. }
```

又绕回了__crypto_alloc_tfm函数，其实现之前已经分析过了，对于一个普通的算法(非模版产生的算法，如md5)，其初始化工作略有不同，在了解其初始化工作之前，需要对一个实际的算法作了解。

顺例说一句，内核的这种抽像管理方式，功能异常地强大，可以想像，它可以抽像更多层的嵌套。所以hmac(xxx)中，xxx不一定就是一个md5之类，可能还是一层形如xxx(xxx)的抽像，理论上，它可以像变形金刚一样。

4.3 小结一下
本节分析了一个算法的tfm是如何生成的，因为算法可以是多层的组装，在生成上层算法的同时，它也要为其所包含的算法分配tfm，这一过程称之为spawn。

http://bbs.chinaunix.net/thread-3627341-1-1.html

分类: linux 内核学习

h13
关注 - 1
粉丝 - 308
+加关注

0 0

« 上一篇： linux内存的使用与page buffer (转)
» 下一篇： ARM的CACHE原理(转)

posted on 2013-03-25 22:24 h13 阅读(4308) 评论(0) 编辑 收藏

相关博文：
· Linux块设备加密之dm-crypt分析
· Linux驱动框架之framebuffer驱动框架
· Linux内核分析（五）----字符设备驱动实现
· MSI中断与Linux实现
· Linux驱动框架之misc类设备驱动框架
» 更多推荐...

最新 IT 新闻:
· 5G时代的短信，能取代我们每天在用的聊天软件吗?
· Chrome OS 开始使用 PWA 替代部分 Android APP
· 外卖巨头DoorDash CEO：疫情是餐饮业30多年来遭遇的最大挑战
· AR试妆、联动牙刷，天猫精灵公布第二代智能美妆镜概念机
· 商家群起而攻美团涨佣"吸血"，官方回应称：我们也赚不到钱
» 更多新闻...