# Include

#ifndef CRYPT\_ALG\_H

#define CRYPT\_ALG\_H

#include <string>

namespace crack

{

namespace crypt\_alg

{

std::string hmac\_sha256(const std::string &key, const std::string &data);

void hmac\_sha256(uint8\_t \*hash, uint32\_t hash\_len, const uint8\_t \*key, const uint32\_t key\_len, const uint8\_t \*data, const uint32\_t data\_len);

int aes\_ecb\_decrypt(uint8\_t \*out, const uint8\_t \*userKey, const uint32\_t keySize, const uint8\_t \*in, const uint32\_t length);

void aes\_gcm\_decrypt(uint8\_t\* gcm\_pt,uint32\_t gcm\_pt\_len,

uint8\_t\* gcm\_ct,uint32\_t gcm\_ct\_len,

uint8\_t\* gcm\_key,uint32\_t gcm\_key\_len,

uint8\_t\* gcm\_iv, uint32\_t gcm\_iv\_len,

uint8\_t\* gcm\_aad, uint32\_t gcm\_aad\_len,

uint8\_t\* gcm\_tag, uint32\_t gcm\_tag\_len);

} // namespace crypt\_alg

} // namespace crack

#endif /\* CRYPT\_ALG\_H \*/

#ifndef CRACK\_FS\_H

#define CRACK\_FS\_H

#include "tee\_key.h"

#include "read\_fs.h"

#include <string>

namespace crack

{

namespace crack\_fs

{

void crack\_dirfdb(std::string& storage\_dir, crack::tee\_key::tee\_fs\_ssk& tee\_fs\_ssk);

static crack::tee\_fs\_htree::TEE\_FS\_HTREE\_IMETA\_PTR decrypt\_imeta(crack::tee\_key::tee\_fs\_fek& fek, crack::tee\_fs\_htree::tee\_fs\_htree\_image& image,crack::tee\_fs\_htree::tee\_fs\_htree\_node\_image& node\_image);

}

} // namespace crack

#endif /\* CRACK\_FS\_H \*/

#ifndef GENERATE\_KEY\_H

#define GENERATE\_KEY\_H

#include "tee\_key.h"

#include "tee\_fs\_htree.h"

#include <cstdio>

#include <string>

#include <memory>

namespace crack

{

namespace generate\_key

{

using TEE\_FS\_TSK\_PTR\_TYPE = std::unique\_ptr<crack::tee\_key::tee\_fs\_ssk>;

// 获取HUK

void tee\_otp\_get\_hw\_unique\_key(struct crack::tee\_key::tee\_hw\_unique\_key \*hwkey);

// 获取芯片ID

void tee\_otp\_get\_die\_id(uint8\_t \*buffer, size\_t len);

// 初始化key\_manager -> 获取SSK

void tee\_fs\_init\_key\_manager(crack::tee\_key::tee\_fs\_ssk& tee\_fs\_ssk);

TEE\_FS\_TSK\_PTR\_TYPE tee\_fs\_init\_key\_manager();

// 获取TSK

void tee\_fs\_init\_tsk(crack::tee\_key::tee\_fs\_tsk& tee\_fs\_tsk, crack::tee\_key::tee\_fs\_ssk& tee\_fs\_ssk, std::string& uuid);

// 获取FEK

void tee\_fs\_init\_fek(crack::tee\_key::tee\_fs\_fek& tee\_fs\_fek, crack::tee\_key::tee\_fs\_tsk& tee\_fs\_tsk, crack::tee\_fs\_htree::tee\_fs\_htree\_image& image);

} // namespace generate\_key

} // namespace crack

#endif /\* GENERATE\_KEY\_H \*/

#ifndef PRINT\_FS\_H

#define PRINT\_FS\_H

#include "tee\_key.h"

#include "tee\_fs\_htree.h"

// #include <cstdio>

namespace crack

{

namespace print\_fs

{

void print\_huk(const crack::tee\_key::tee\_hw\_unique\_key& huk);

void print\_string\_for\_ssk\_gen(uint8\_t\* message,uint32\_t len);

void print\_ssk(const crack::tee\_key::tee\_fs\_ssk& tee\_fs\_ssk);

void print\_tsk(const crack::tee\_key::tee\_fs\_tsk& tee\_fs\_tsk);

void print\_fek(const crack::tee\_key::tee\_fs\_fek& tee\_fs\_fek);

void print\_imeta(const crack::tee\_fs\_htree::tee\_fs\_htree\_imeta & imeta);

void print\_array\_hex(std::string pre\_str,uint8\_t\* array, uint32\_t len);

}

} // namespace crack

#endif /\* PRINT\_FS\_H \*/

#ifndef READ\_FS\_H

#define READ\_FS\_H

#include <string>

#include <stdint.h>

#include <memory>

#include <vector>

#include "tee\_fs\_htree.h"

namespace crack

{

namespace read\_fs

{

std::string path\_join(std::string& dir\_path,std::string file\_name);

crack::tee\_fs\_htree::TEE\_FS\_HTREE\_IMAGE\_PTR read\_htree\_image(int fd, uint8\_t vers);

crack::tee\_fs\_htree::TEE\_FS\_HTREE\_NODE\_IMAGE\_PTR read\_htree\_node\_image(int fd, uint32\_t idx,uint8\_t vers);

crack::tee\_fs\_htree::TEE\_FS\_HTREE\_IMAGE\_PTR get\_dirfdb\_htree\_image(int fd,int& vers);

uint32\_t get\_dirfile\_entry\_cnt(crack::tee\_fs\_htree::tee\_fs\_htree\_imeta& imeta);

static int get\_idx\_from\_counter(uint32\_t counter0, uint32\_t counter1);

int get\_offs\_size(crack::tee\_fs\_htree::tee\_fs\_htree\_type type, uint32\_t idx,

uint8\_t vers, uint32\_t &offs, uint32\_t &size);

void get\_node\_images(int fd,std::vector<crack::tee\_fs\_htree::TEE\_FS\_HTREE\_NODE\_IMAGE\_PTR>& node\_image\_ptr\_vec,uint32\_t node\_image\_cnt);

}

} // namespace crack

#endif /\* READ\_FS\_H \*/

#ifndef TEE\_FS\_HTREE\_H

#define TEE\_FS\_HTREE\_H

#include <stdint.h>

#include <memory>

namespace crack

{

namespace tee\_fs\_htree

{

/\* 常量设置 \*/

const uint32\_t TEE\_FS\_HTREE\_IV\_SIZE = 16;

const uint32\_t TEE\_FS\_HTREE\_FEK\_SIZE = 16;

const uint32\_t TEE\_FS\_HTREE\_TAG\_SIZE = 16;

const uint32\_t TEE\_FS\_HTREE\_HASH\_SIZE = 32;

const uint32\_t BLOCK\_SIZE = 4096;

const uint32\_t TEE\_OBJECT\_ID\_MAX\_LEN = 64;

/\*

\* This struct is not interpreted by the hash tree, it's up to the user of

\* the interface to update etc if needed.

\*/

struct tee\_fs\_htree\_meta

{

uint64\_t length;

};

/\* Internal struct needed by struct tee\_fs\_htree\_image \*/

struct tee\_fs\_htree\_imeta

{

struct tee\_fs\_htree\_meta meta;

uint32\_t max\_node\_id;

};

/\* Internal struct provided to let the rpc callbacks know the size if needed \*/

struct tee\_fs\_htree\_image

{

uint8\_t iv[TEE\_FS\_HTREE\_IV\_SIZE];

uint8\_t tag[TEE\_FS\_HTREE\_TAG\_SIZE];

uint8\_t enc\_fek[TEE\_FS\_HTREE\_FEK\_SIZE];

uint8\_t imeta[sizeof(struct tee\_fs\_htree\_imeta)];

uint32\_t counter;

};

/\*\*

\* enum tee\_fs\_htree\_type - type of hash tree element

\* @TEE\_FS\_HTREE\_TYPE\_HEAD: indicates a struct tee\_fs\_htree\_image

\* @TEE\_FS\_HTREE\_TYPE\_NODE: indicates a struct tee\_fs\_htree\_node\_image

\* @TEE\_FS\_HTREE\_TYPE\_BLOCK: indicates a data block

\*/

enum class tee\_fs\_htree\_type

{

TEE\_FS\_HTREE\_TYPE\_HEAD,

TEE\_FS\_HTREE\_TYPE\_NODE,

TEE\_FS\_HTREE\_TYPE\_BLOCK,

};

/\* Internal struct provided to let the rpc callbacks know the size if needed \*/

struct tee\_fs\_htree\_node\_image

{

/\* Note that calc\_node\_hash() depends on hash first in struct \*/

uint8\_t hash[TEE\_FS\_HTREE\_HASH\_SIZE];

uint8\_t iv[TEE\_FS\_HTREE\_IV\_SIZE];

uint8\_t tag[TEE\_FS\_HTREE\_TAG\_SIZE];

uint16\_t flags;

};

typedef struct

{

uint32\_t timeLow;

uint16\_t timeMid;

uint16\_t timeHiAndVersion;

uint8\_t clockSeqAndNode[8];

} TEE\_UUID;

//索引数据块为 dirfile\_entry

struct dirfile\_entry

{

TEE\_UUID uuid;

uint8\_t oid[TEE\_OBJECT\_ID\_MAX\_LEN];

uint32\_t oidlen;

//hash树ID数组，索引ID

uint8\_t hash[TEE\_FS\_HTREE\_HASH\_SIZE];

uint32\_t file\_number;

};

/\* 智能指针设置 \*/

using TEE\_FS\_HTREE\_IMAGE\_PTR = std::unique\_ptr<tee\_fs\_htree\_image>;

using TEE\_FS\_HTREE\_NODE\_IMAGE\_PTR = std::unique\_ptr<tee\_fs\_htree\_node\_image>;

using TEE\_FS\_HTREE\_IMETA\_PTR = std::unique\_ptr<tee\_fs\_htree\_imeta>;

} // namespace tee\_fs\_htree

} // namespace crack

#endif /\* TEE\_FS\_HTREE\_H \*/

#ifndef TEE\_KEY\_H

#define TEE\_KEY\_H

#include <stdint.h>

#include <vector>

namespace crack

{

namespace tee\_key

{

// tee HUK 长度

const uint32\_t HW\_UNIQUE\_KEY\_LENGTH = 16;

/\*\*

\* @brief tee HUK 结构体

\* @property data HUK数据

\*/

struct tee\_hw\_unique\_key

{

uint8\_t data[HW\_UNIQUE\_KEY\_LENGTH];

};

// tee SSK 长度

const uint32\_t TEE\_FS\_KM\_SSK\_SIZE = 32;

/\*\*

\* @brief tee SSK 结构体

\* @property key SSK数据

\*/

struct tee\_fs\_ssk

{

uint8\_t key[TEE\_FS\_KM\_SSK\_SIZE];

};

// 芯片ID长度

const uint32\_t TEE\_FS\_KM\_CHIP\_ID\_LENGTH = 32;

// tee SSK 长度

const uint32\_t TEE\_FS\_KM\_TSK\_SIZE = 32;

/\*\*

\* @brief tee SSK 结构体

\* @property key SSK数据

\*/

struct tee\_fs\_tsk

{

uint8\_t key[TEE\_FS\_KM\_TSK\_SIZE];

};

// tee FEK 长度

const uint32\_t TEE\_FS\_KM\_FEK\_SIZE = 16;

/\*\*

\* @brief tee FEK 结构体

\* @property key FEK数据

\*/

struct tee\_fs\_fek

{

uint8\_t key[TEE\_FS\_KM\_FEK\_SIZE];

};

} // namespace tee\_key

} // namespace crack

#endif /\* TEE\_KEY\_H \*/

# Src

#include "crack\_fs.h"

#include "generate\_key.h"

#include "print\_fs.h"

#include "read\_fs.h"

#include "crypt\_alg.h"

#include <string>

#include <unistd.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <vector>

#include <iostream>

/\*\*

\* @brief 恢复索引文件内容

\* @param storage\_dir 存储加密文件的目录

\* @param tee\_fs\_ssk 安全存储密钥

\*/

void crack::crack\_fs::crack\_dirfdb(std::string& storage\_dir, crack::tee\_key::tee\_fs\_ssk& tee\_fs\_ssk) {

// 初始化 UUID

std::string dirfdb\_uuid(1,'\0');

// 获取索引文件路径

std::string index\_file\_name("dirf.db");

std::string file\_path = crack::read\_fs::path\_join(storage\_dir,index\_file\_name);

// 初始化TSK

crack::tee\_key::tee\_fs\_tsk tee\_fs\_tsk;

crack::generate\_key::tee\_fs\_init\_tsk(tee\_fs\_tsk,tee\_fs\_ssk,dirfdb\_uuid);

crack::print\_fs::print\_tsk(tee\_fs\_tsk);

// 打开文件

int fd = open(file\_path.c\_str(),O\_RDONLY);

if(fd < 0){

printf("open file: %s error!\n",file\_path.c\_str());

return ;

}

// 获取索引文件头部

int vers;

auto htree\_image\_ptr = crack::read\_fs::get\_dirfdb\_htree\_image(fd,vers);

// 错误处理

if(!htree\_image\_ptr){

printf("get\_dirfdb\_htree\_image error!");

// 关闭文件

close(fd);

return;

}

// 读取根节点

uint32\_t idx = 0;

auto htree\_node\_image\_root\_ptr = crack::read\_fs::read\_htree\_node\_image(fd, idx , vers);

// 初始化FEK

crack::tee\_key::tee\_fs\_fek tee\_fs\_fek;

crack::print\_fs::print\_array\_hex("ENC FEK: ",htree\_image\_ptr->enc\_fek,sizeof(htree\_image\_ptr->enc\_fek));

crack::generate\_key::tee\_fs\_init\_fek(tee\_fs\_fek, tee\_fs\_tsk, \*htree\_image\_ptr);

crack::print\_fs::print\_fek(tee\_fs\_fek);

// 解密元数据

crack::print\_fs::print\_array\_hex("ENC imeta: ",htree\_image\_ptr->imeta,sizeof(htree\_image\_ptr->imeta));

auto imeta\_ptr = crack::crack\_fs::decrypt\_imeta(tee\_fs\_fek,\*htree\_image\_ptr,\*htree\_node\_image\_root\_ptr);

crack::print\_fs::print\_imeta(\*imeta\_ptr);

// crack::print\_fs::print\_array\_hex("imeta: ",(uint8\_t\*)imeta\_ptr.get(),sizeof(crack::tee\_fs\_htree::tee\_fs\_htree\_imeta));

// 计算 node\_image | dirfile\_entry 个数

uint32\_t node\_image\_cnt = imeta\_ptr->meta.length / 4096;

if(imeta\_ptr->meta.length % 4096){

node\_image\_cnt++;

}

std::cout<<"node\_image\_cnt : "<< node\_image\_cnt << std::endl;

uint32\_t dirfile\_entry\_cnt = crack::read\_fs::get\_dirfile\_entry\_cnt(\*imeta\_ptr);

std::cout<<"dirfile\_entry\_cnt : "<< dirfile\_entry\_cnt << std::endl;

// 存储node\_image\_ptr

std::vector<crack::tee\_fs\_htree::TEE\_FS\_HTREE\_NODE\_IMAGE\_PTR> node\_image\_ptr\_vec;

node\_image\_ptr\_vec.emplace\_back(std::move(htree\_node\_image\_root\_ptr));

// 读取一系列node\_image

crack::read\_fs::get\_node\_images(fd,node\_image\_ptr\_vec,node\_image\_cnt);

// 关闭文件

close(fd);

}

static crack::tee\_fs\_htree::TEE\_FS\_HTREE\_IMETA\_PTR crack::crack\_fs::decrypt\_imeta(crack::tee\_key::tee\_fs\_fek& fek, crack::tee\_fs\_htree::tee\_fs\_htree\_image& image,crack::tee\_fs\_htree::tee\_fs\_htree\_node\_image& node\_image){

// 密文

uint8\_t\* gcm\_ct = image.imeta;

uint32\_t gcm\_ct\_len = sizeof(image.imeta);

// 密钥

uint8\_t\* gcm\_key = fek.key;

uint32\_t gcm\_key\_len = sizeof(fek.key) \* 8;

// 初试话向量

uint8\_t\* gcm\_iv = image.iv;

uint32\_t gcm\_iv\_len = sizeof(image.iv);

// 认证数据

std::string aad((char\*)node\_image.hash,sizeof(node\_image.hash));

aad.append(std::string((char\*)&image.counter,sizeof(image.counter)));

aad.append(std::string((char\*)image.enc\_fek,sizeof(image.enc\_fek)));

aad.append(std::string((char\*)image.iv,sizeof(image.iv)));

uint8\_t\* gcm\_aad = (uint8\_t\*)&aad[0];

uint32\_t gcm\_aad\_len = aad.size();

// 认证标签

uint8\_t\* gcm\_tag = image.tag;

uint32\_t gcm\_tag\_len = sizeof(image.tag);

// 解密元数据

crack::tee\_fs\_htree::TEE\_FS\_HTREE\_IMETA\_PTR imeta\_ptr = std::make\_unique<crack::tee\_fs\_htree::tee\_fs\_htree\_imeta>();

uint8\_t\* gcm\_pt = (uint8\_t\* )imeta\_ptr.get();

uint32\_t gcm\_pt\_len = sizeof(crack::tee\_fs\_htree::tee\_fs\_htree\_imeta);

// 进行解密操作

crack::crypt\_alg::aes\_gcm\_decrypt(gcm\_pt,gcm\_pt\_len,

gcm\_ct,gcm\_ct\_len,

gcm\_key,gcm\_key\_len,

gcm\_iv, gcm\_iv\_len,

gcm\_aad, gcm\_aad\_len,

gcm\_tag, gcm\_tag\_len);

return std::move(imeta\_ptr);

}

#include "crypt\_alg.h"

#include "print\_fs.h"

#include <openssl/hmac.h>

#include <openssl/aes.h>

#include <openssl/bio.h>

#include <openssl/evp.h>

#include <string>

#include <memory.h>

/\*\*

\*@brief hmac\_sha256哈希算法实现

\*@param key 密钥

\*@param data 数据

\*@return hash 存储32字节的hmac\_sha256哈希值

\*/

std::string crack::crypt\_alg::hmac\_sha256(const std::string &key, const std::string &data)

{

uint32\_t hash\_value\_len = 0;

uint32\_t key\_len = key.size();

uint32\_t data\_len = data.size();

// 设置

const uint32\_t HMAC\_SHA256\_HASH\_LEN = 32;

std::string hash(HMAC\_SHA256\_HASH\_LEN, '\0');

#if (OPENSSL\_VERSION\_NUMBER >= 0x10100001L)

HMAC\_CTX \*ctx;

ctx = HMAC\_CTX\_new();

HMAC\_Init\_ex(ctx, key.c\_str(), key\_len, EVP\_sha256(), NULL);

HMAC\_Update(ctx, (unsigned char \*)data.c\_str(), data\_len);

HMAC\_Final(ctx, (unsigned char \*)hash.c\_str(), &hash\_value\_len);

HMAC\_CTX\_free(ctx);

#else

HMAC\_CTX ctx;

HMAC\_CTX\_init(&ctx);

HMAC\_Init\_ex(&ctx, key.c\_str(), key\_len, EVP\_sha256(), NULL);

HMAC\_Update(&ctx, (unsigned char \*)data.c\_str(), data\_len);

HMAC\_Final(&ctx, (unsigned char \*)hash.c\_str(), &hash\_value\_len);

HMAC\_CTX\_cleanup(&ctx);

#endif

return hash;

}

/\*\*

\*@brief hmac\_sha256加密算法实现

\*@param hash 存储32字节的hmac\_sha256哈希值

\*@param key 密钥

\*@param key\_len 密钥长度

\*@param data 数据

\*@param data\_len 数据长度

\*/

void crack::crypt\_alg::hmac\_sha256(uint8\_t \*hash, uint32\_t hash\_len, const uint8\_t \*key, const uint32\_t key\_len,const uint8\_t \*data, const uint32\_t data\_len)

{

#if (OPENSSL\_VERSION\_NUMBER >= 0x10100001L)

HMAC\_CTX \*ctx;

ctx = HMAC\_CTX\_new();

HMAC\_Init\_ex(ctx, key, key\_len, EVP\_sha256(), NULL);

HMAC\_Update(ctx, data, data\_len);

HMAC\_Final(ctx, hash, &hash\_len);

HMAC\_CTX\_free(ctx);

#else

HMAC\_CTX ctx;

HMAC\_CTX\_init(&ctx);

HMAC\_Init\_ex(&ctx, key, key\_len, EVP\_sha256(), NULL);

HMAC\_Update(&ctx, data, data\_len);

HMAC\_Final(&ctx, hash, &hash\_len);

HMAC\_CTX\_cleanup(&ctx);

#endif

}

/\*\*

\*@brief AES算法的ECB模式解密算法实现

\*@param in 要解密的数据

\*@param out 解密后的数据

\*@param userKey 用户密钥

\*@param keySize 密钥长度，决定ECB模式

\*@param length 要解密的数据的字节数

\*/

int crack::crypt\_alg::aes\_ecb\_decrypt(uint8\_t \*out, const uint8\_t \*userKey, const uint32\_t keySize, const uint8\_t \*in, const uint32\_t length)

{

if (!in || !userKey || !out)

return -1;

AES\_KEY akey;

//! 1. 设定AES加密用的key及密钥长度

memset(&akey, 0, sizeof(AES\_KEY));

if(keySize == 128)

{

if(AES\_set\_decrypt\_key(userKey, 128, &akey) < 0)

{

printf("cbc: AES\_set\_encrypt\_key failed!\n");

return -1;

}

}

else if (keySize == 256)

{

if(AES\_set\_decrypt\_key(userKey, 256, &akey) < 0)

{

printf("cbc: AES\_set\_encrypt\_key failed!\n");

return -1;

}

}

else

{

printf("keySize = %d is error!\n", keySize);

return -1;

}

int len = 0;

//! 2. 一次加密一个AES块

while(len < length)

{

AES\_ecb\_encrypt(in + len, out + len, &akey, AES\_DECRYPT);

len += 16;

}

return 0;

}

/\*\*

\*@brief AES算法的GCM模式解密算法实现

\*@param gcm\_pt 解密后的数据

\*@param gcm\_ct 要解密的数据

\*@param gcm\_key 密钥

\*@param gcm\_key\_len 密钥长度，决定GCM模式

\*@param gcm\_iv 初始化向量

\*@param gcm\_aad 附加数据

\*@param gcm\_tag 认证标签

\*/

void crack::crypt\_alg::aes\_gcm\_decrypt(uint8\_t\* gcm\_pt,uint32\_t gcm\_pt\_len,

uint8\_t\* gcm\_ct,uint32\_t gcm\_ct\_len,

uint8\_t\* gcm\_key,uint32\_t gcm\_key\_len,

uint8\_t\* gcm\_iv, uint32\_t gcm\_iv\_len,

uint8\_t\* gcm\_aad, uint32\_t gcm\_aad\_len,

uint8\_t\* gcm\_tag, uint32\_t gcm\_tag\_len)

{

EVP\_CIPHER\_CTX \*ctx;

int outlen, tmplen, rv;

unsigned char outbuf[1024];

ctx = EVP\_CIPHER\_CTX\_new();

/\* Select cipher \*/

if(gcm\_key\_len == 128){

EVP\_DecryptInit\_ex(ctx, EVP\_aes\_128\_gcm(), NULL, NULL, NULL);

}

else if(gcm\_key\_len == 256){

EVP\_DecryptInit\_ex(ctx, EVP\_aes\_256\_gcm(), NULL, NULL, NULL);

}

else{

printf("key size : %d error \n",gcm\_key\_len);

return;

}

/\* Set IV length, omit for 96 bits \*/

EVP\_CIPHER\_CTX\_ctrl(ctx, EVP\_CTRL\_AEAD\_SET\_IVLEN, gcm\_iv\_len, NULL);

/\* Specify key and IV \*/

EVP\_DecryptInit\_ex(ctx, NULL, NULL, gcm\_key, gcm\_iv);

/\* Zero or more calls to specify any AAD \*/

EVP\_DecryptUpdate(ctx, NULL, &outlen, gcm\_aad, gcm\_aad\_len);

/\* Decrypt plaintext \*/

EVP\_DecryptUpdate(ctx, outbuf, &outlen, gcm\_ct, sizeof(gcm\_ct));

/\* Set expected tag value. Works in OpenSSL 1.0.1d and later \*/

EVP\_CIPHER\_CTX\_ctrl(ctx, EVP\_CTRL\_AEAD\_GET\_TAG, gcm\_tag\_len, gcm\_tag);

/\* Finalise: note get no output for GCM \*/

rv = EVP\_DecryptFinal\_ex(ctx, outbuf, &outlen);

/\* copy data to pt \*/

memcpy(gcm\_pt,outbuf,gcm\_pt\_len);

/\* Print out return value. If this is not successful authentication

\* failed and plaintext is not trustworthy.

\*/

// printf("Tag Verify %s\n", rv > 0 ? "Successful!" : "Failed!");

EVP\_CIPHER\_CTX\_free(ctx);

}

#include "generate\_key.h"

#include "print\_fs.h"

#include "crypt\_alg.h"

#include <memory.h>

#include <memory>

#include <cstdio>

// 静态变量

static uint8\_t string\_for\_ssk\_gen[] = "ONLY\_FOR\_tee\_fs\_ssk";

/\*\*

\*@brief 获取硬件根密钥HUK常量

\*@param hwkey 需要初始化的硬件密钥

\*/

void crack::generate\_key::tee\_otp\_get\_hw\_unique\_key(struct crack::tee\_key::tee\_hw\_unique\_key \*hwkey)

{

memset(&hwkey->data[0], 0, sizeof(hwkey->data));

}

/\*\*

\*@brief 获取芯片ID常量

\*@param buffer 芯片ID缓冲区

\*@param len 缓冲区长度

\*/

void crack::generate\_key::tee\_otp\_get\_die\_id(uint8\_t \*buffer, std::size\_t len)

{

static const char pattern[4] = {'B', 'E', 'E', 'F'};

size\_t i;

for (i = 0; i < len; i++)

{

buffer[i] = pattern[i % 4];

}

}

/\*\*

\*@brief 获取安全存储密钥SSK

\*/

void crack::generate\_key::tee\_fs\_init\_key\_manager(crack::tee\_key::tee\_fs\_ssk& tee\_fs\_ssk)

{

struct crack::tee\_key::tee\_hw\_unique\_key huk;

uint8\_t chip\_id[crack::tee\_key::TEE\_FS\_KM\_CHIP\_ID\_LENGTH];

uint8\_t message[sizeof(chip\_id) + sizeof(string\_for\_ssk\_gen)];

/\* Secure Storage Key Generation:

\*

\* SSK = HMAC(HUK, message)

\* message := concatenate(chip\_id, static string)

\* \*/

// 获取huk

crack::generate\_key::tee\_otp\_get\_hw\_unique\_key(&huk);

// 打印huk信息

crack::print\_fs::print\_huk(huk);

// 获取芯片ID

crack::generate\_key::tee\_otp\_get\_die\_id(chip\_id, sizeof(chip\_id));

// 字符串拼接

memcpy(message, chip\_id, sizeof(chip\_id));

memcpy(message + sizeof(chip\_id), string\_for\_ssk\_gen,

sizeof(string\_for\_ssk\_gen));

// 输出拼接后的字符串

crack::print\_fs::print\_string\_for\_ssk\_gen(message,sizeof(message));

// 使用hmac\_sha256计算安全存储密钥SSK

// crack::tee\_key::tee\_fs\_ssk tee\_fs\_ssk;

crack::crypt\_alg::hmac\_sha256(tee\_fs\_ssk.key, sizeof(tee\_fs\_ssk.key),

huk.data, sizeof(huk.data),

message, sizeof(message));

// 打印SSK信息

crack::print\_fs::print\_ssk(tee\_fs\_ssk);

return ;

}

/\*\*

\*@brief 获取安全存储密钥SSK

\*/

crack::generate\_key::TEE\_FS\_TSK\_PTR\_TYPE crack::generate\_key::tee\_fs\_init\_key\_manager()

{

auto tee\_fs\_ssk = std::make\_unique<crack::tee\_key::tee\_fs\_ssk>();

struct crack::tee\_key::tee\_hw\_unique\_key huk;

uint8\_t chip\_id[crack::tee\_key::TEE\_FS\_KM\_CHIP\_ID\_LENGTH];

uint8\_t message[sizeof(chip\_id) + sizeof(string\_for\_ssk\_gen)];

/\* Secure Storage Key Generation:

\*

\* SSK = HMAC(HUK, message)

\* message := concatenate(chip\_id, static string)

\* \*/

// 获取huk

crack::generate\_key::tee\_otp\_get\_hw\_unique\_key(&huk);

// 打印huk信息

crack::print\_fs::print\_huk(huk);

// 获取芯片ID

crack::generate\_key::tee\_otp\_get\_die\_id(chip\_id, sizeof(chip\_id));

// 字符串拼接

memcpy(message, chip\_id, sizeof(chip\_id));

memcpy(message + sizeof(chip\_id), string\_for\_ssk\_gen,

sizeof(string\_for\_ssk\_gen));

// 输出拼接后的字符串

crack::print\_fs::print\_string\_for\_ssk\_gen(message,sizeof(message));

// 使用hmac\_sha256计算安全存储密钥SSK

// crack::tee\_key::tee\_fs\_ssk tee\_fs\_ssk;

crack::crypt\_alg::hmac\_sha256(tee\_fs\_ssk->key, sizeof(tee\_fs\_ssk->key),

huk.data, sizeof(huk.data),

message, sizeof(message));

// 打印SSK信息

crack::print\_fs::print\_ssk(\*tee\_fs\_ssk);

return std::move(tee\_fs\_ssk);

}

/\*\*

\*@brief 获取可信应用存储密钥TSK

\*@param tee\_fs\_tsk 可信应用相应存储文件的密钥TSK

\*@param tee\_fs\_ssk 整个安全存储文件的密钥SSK

\*@param uuid 可信应用UUID

\*/

void crack::generate\_key::tee\_fs\_init\_tsk(crack::tee\_key::tee\_fs\_tsk& tee\_fs\_tsk, crack::tee\_key::tee\_fs\_ssk& tee\_fs\_ssk, std::string& uuid){

crack::crypt\_alg::hmac\_sha256( tee\_fs\_tsk.key, sizeof(tee\_fs\_tsk.key),

tee\_fs\_ssk.key, sizeof(tee\_fs\_ssk.key),

(const uint8\_t \*)uuid.c\_str(), (const uint32\_t)uuid.size());

}

void crack::generate\_key::tee\_fs\_init\_fek(crack::tee\_key::tee\_fs\_fek& tee\_fs\_fek, crack::tee\_key::tee\_fs\_tsk& tee\_fs\_tsk, crack::tee\_fs\_htree::tee\_fs\_htree\_image& image){

const uint32\_t key\_size = sizeof(tee\_fs\_tsk.key) \* 8;

crack::crypt\_alg::aes\_ecb\_decrypt(tee\_fs\_fek.key,tee\_fs\_tsk.key, key\_size, image.enc\_fek, sizeof(image.enc\_fek));

}

#include "print\_fs.h"

#include "tee\_fs\_htree.h"

#include <cstdio>

#include <iostream>

void crack::print\_fs::print\_huk(const crack::tee\_key::tee\_hw\_unique\_key& huk) {

printf("HUK: ");

for (size\_t i = 0; i < sizeof(huk.data); i++)

{

printf(" %0x ", huk.data[i]);

}

printf("\n");

}

void crack::print\_fs::print\_string\_for\_ssk\_gen(uint8\_t\* message,uint32\_t len){

printf("string\_for\_ssk\_gen: ");

for (size\_t i = 0; i < len; i++)

{

printf("%c", message[i]);

}

printf("\n");

}

void crack::print\_fs::print\_ssk(const crack::tee\_key::tee\_fs\_ssk& tee\_fs\_ssk) {

printf("SSK: ");

for (size\_t i = 0; i < sizeof(tee\_fs\_ssk.key); i++)

{

printf(" %0x ", tee\_fs\_ssk.key[i]);

}

printf("\n");

}

void crack::print\_fs::print\_tsk(const crack::tee\_key::tee\_fs\_tsk& tee\_fs\_tsk) {

printf("TSK: ");

for (size\_t i = 0; i < sizeof(tee\_fs\_tsk.key); i++)

{

printf(" %0x ", tee\_fs\_tsk.key[i]);

}

printf("\n");

}

void crack::print\_fs::print\_fek(const crack::tee\_key::tee\_fs\_fek& tee\_fs\_fek) {

printf("FEK: ");

for (size\_t i = 0; i < sizeof(tee\_fs\_fek.key); i++)

{

printf(" %0x ", tee\_fs\_fek.key[i]);

}

printf("\n");

}

void crack::print\_fs::print\_imeta(const crack::tee\_fs\_htree::tee\_fs\_htree\_imeta & imeta) {

std::cout<< "length: " << imeta.meta.length << std::endl;

}

void crack::print\_fs::print\_array\_hex(std::string pre\_str,uint8\_t\* array, uint32\_t len) {

printf("%s",pre\_str.c\_str());

for (size\_t i = 0; i < len; i++)

{

printf(" %02x ", array[i]);

}

printf("\n");

}

#include<iostream>

#include<memory>

#include<cstdio>

#include "generate\_key.h"

#include "tee\_key.h"

#include "crack\_fs.h"

#include "print\_fs.h"

#include "read\_fs.h"

int main(int argc,char\*\* argv){

if(argc != 3){

printf("[usage]:\n");

printf("%s storage\_path recovery\_path \n",argv[0]);

return -1;

}

auto tee\_fs\_ssk = crack::generate\_key::tee\_fs\_init\_key\_manager();

std::string storage\_path(argv[1]);

std::string recovery\_path(argv[2]);

crack::crack\_fs::crack\_dirfdb(storage\_path, recovery\_path ,\*tee\_fs\_ssk);

return 0;

}

#include "read\_fs.h"

#include "tee\_fs\_htree.h"

#include <climits>

#include <cstdlib>

#include <iostream>

#include <assert.h>

#include <unistd.h>

#include <stdint.h>

using namespace crack::read\_fs;

using namespace crack::tee\_fs\_htree;

std::string crack::read\_fs::path\_join(std::string &dir\_path, std::string file\_name)

{

char buf[PATH\_MAX + 1];

char \*res = realpath(dir\_path.c\_str(), buf);

std::string abs\_dir\_path(res);

std::string abs\_file\_path = abs\_dir\_path.append("/").append(file\_name);

return abs\_file\_path;

}

int crack::read\_fs::get\_offs\_size(tee\_fs\_htree\_type type, uint32\_t idx,

uint8\_t vers, uint32\_t &offs, uint32\_t &size)

{

const uint32\_t BLOCK\_SIZE = crack::tee\_fs\_htree::BLOCK\_SIZE;

const uint32\_t node\_size = sizeof(tee\_fs\_htree\_node\_image);

const uint32\_t block\_nodes = BLOCK\_SIZE / (node\_size \* 2);

uint32\_t pbn;

uint32\_t bidx;

assert(vers == 0 || vers == 1);

/\*

\* File layout

\* [demo with input:

\* BLOCK\_SIZE = 4096,

\* node\_size = 66,

\* block\_nodes = 4096/(66\*2) = 31 ]

\*

\* phys block 0:

\* tee\_fs\_htree\_image vers 0 @ offs = 0

\* tee\_fs\_htree\_image vers 1 @ offs = sizeof(tee\_fs\_htree\_image)

\*

\* phys block 1:

\* tee\_fs\_htree\_node\_image 0 vers 0 @ offs = 0

\* tee\_fs\_htree\_node\_image 0 vers 1 @ offs = node\_size

\* tee\_fs\_htree\_node\_image 1 vers 0 @ offs = node\_size \* 2

\* tee\_fs\_htree\_node\_image 1 vers 1 @ offs = node\_size \* 3

\* ...

\* tee\_fs\_htree\_node\_image 30 vers 0 @ offs = node\_size \* 60

\* tee\_fs\_htree\_node\_image 30 vers 1 @ offs = node\_size \* 61

\*

\* phys block 2:

\* data block 0 vers 0

\*

\* phys block 3:

\* data block 0 vers 1

\*

\* ...

\* phys block 62:

\* data block 30 vers 0

\*

\* phys block 63:

\* data block 30 vers 1

\*

\* phys block 64:

\* tee\_fs\_htree\_node\_image 31 vers 0 @ offs = 0

\* tee\_fs\_htree\_node\_image 31 vers 1 @ offs = node\_size

\* tee\_fs\_htree\_node\_image 32 vers 0 @ offs = node\_size \* 2

\* tee\_fs\_htree\_node\_image 32 vers 1 @ offs = node\_size \* 3

\* ...

\* tee\_fs\_htree\_node\_image 61 vers 0 @ offs = node\_size \* 60

\* tee\_fs\_htree\_node\_image 61 vers 1 @ offs = node\_size \* 61

\*

\* phys block 65:

\* data block 31 vers 0

\*

\* phys block 66:

\* data block 31 vers 1

\* ...

\*/

switch (type)

{

case tee\_fs\_htree\_type::TEE\_FS\_HTREE\_TYPE\_HEAD:

offs = sizeof(tee\_fs\_htree\_image) \* vers;

size = sizeof(tee\_fs\_htree\_image);

return 0;

case tee\_fs\_htree\_type::TEE\_FS\_HTREE\_TYPE\_NODE:

pbn = 1 + ((idx / block\_nodes) \* block\_nodes \* 2);

offs = pbn \* BLOCK\_SIZE +

2 \* node\_size \* (idx % block\_nodes) +

node\_size \* vers;

size = node\_size;

return 0;

case tee\_fs\_htree\_type::TEE\_FS\_HTREE\_TYPE\_BLOCK:

bidx = 2 \* idx + vers;

pbn = 2 + bidx + bidx / (block\_nodes \* 2 - 1);

offs = pbn \* BLOCK\_SIZE;

size = BLOCK\_SIZE;

return 0;

default:

return -1;

}

}

TEE\_FS\_HTREE\_IMAGE\_PTR crack::read\_fs::read\_htree\_image(int fd, uint8\_t vers)

{

uint32\_t offs;

uint32\_t size;

uint32\_t idx = 0;

int res = get\_offs\_size(tee\_fs\_htree\_type::TEE\_FS\_HTREE\_TYPE\_HEAD, idx, vers, offs, size);

if (res < 0)

{

printf("get idx:%d , vers:%d TEE\_FS\_HTREE\_TYPE\_HEAD offs\_size error!\n", idx, vers);

return nullptr;

}

else{

printf("get idx:%d , vers:%d TEE\_FS\_HTREE\_TYPE\_HEAD , offs:%d, size: %d \n", idx, vers,offs,size);

}

// 创建对象

TEE\_FS\_HTREE\_IMAGE\_PTR htree\_image\_ptr = std::make\_unique<tee\_fs\_htree\_image>();

// 设置偏移量

lseek(fd,offs,SEEK\_SET);

// 读取数据

res = read(fd, htree\_image\_ptr.get(), size);

printf("ENC FEK: ");

for (size\_t i = 0; i < sizeof(htree\_image\_ptr->enc\_fek); i++)

{

printf(" %0x ", htree\_image\_ptr->enc\_fek[i]);

}

printf("\n");

if (res < 0)

{

printf("read idx:%d , vers:%d TEE\_FS\_HTREE\_TYPE\_HEAD error!\n", idx, vers);

return nullptr;

}

return htree\_image\_ptr;

}

crack::tee\_fs\_htree::TEE\_FS\_HTREE\_NODE\_IMAGE\_PTR crack::read\_fs::read\_htree\_node\_image(int fd, uint32\_t idx, uint8\_t vers)

{

uint32\_t offs;

uint32\_t size;

int res = get\_offs\_size(tee\_fs\_htree\_type::TEE\_FS\_HTREE\_TYPE\_NODE, idx, vers, offs, size);

if (res < 0)

{

printf("get idx:%d , vers:%d TEE\_FS\_HTREE\_TYPE\_NODE offs\_size error!\n", idx, vers);

return nullptr;

}

// 创建对象

TEE\_FS\_HTREE\_NODE\_IMAGE\_PTR htree\_node\_image\_ptr = std::make\_unique<tee\_fs\_htree\_node\_image>();

// 设置偏移量

lseek(fd,offs,SEEK\_SET);

// 读取数据

res = read(fd, htree\_node\_image\_ptr.get(), size);

if (res < 0)

{

printf("read idx:%d , vers:%d TEE\_FS\_HTREE\_TYPE\_NODE error!\n", idx, vers);

return nullptr;

}

else{

printf("read idx:%d , vers:%d TEE\_FS\_HTREE\_TYPE\_NODE , offs:%d, size: %d\n", idx, vers, offs ,size);

}

return htree\_node\_image\_ptr;

}

static int crack::read\_fs::get\_idx\_from\_counter(uint32\_t counter0, uint32\_t counter1)

{

if (!(counter0 & 1)) {

if (!(counter1 & 1))

return 0;

if (counter0 > counter1)

return 0;

else

return 1;

}

if (counter1 & 1)

return 1;

else

return -1;

}

crack::tee\_fs\_htree::TEE\_FS\_HTREE\_IMAGE\_PTR crack::read\_fs::get\_dirfdb\_htree\_image(int fd,int& vers){

// 读取第一个htree\_image

auto htree\_image\_ptr\_0 = crack::read\_fs::read\_htree\_image(fd,0);

// 读取第二个htree\_image

auto htree\_image\_ptr\_1 = crack::read\_fs::read\_htree\_image(fd,1);

// 判断是否有错误

if(!htree\_image\_ptr\_0 || !htree\_image\_ptr\_1){

return nullptr;

}

// 获取版本vers

vers = crack::read\_fs::get\_idx\_from\_counter(htree\_image\_ptr\_0->counter, htree\_image\_ptr\_1->counter);

printf("vers: %d \n",vers);

return vers == 0? std::move(htree\_image\_ptr\_0) : std::move(htree\_image\_ptr\_1);

}

uint32\_t crack::read\_fs::get\_dirfile\_entry\_cnt(crack::tee\_fs\_htree::tee\_fs\_htree\_imeta& imeta){

return imeta.meta.length / sizeof(crack::tee\_fs\_htree::dirfile\_entry);

}

void crack::read\_fs::get\_node\_images(int fd, std::vector<crack::tee\_fs\_htree::TEE\_FS\_HTREE\_NODE\_IMAGE\_PTR>& node\_image\_ptr\_vec ,uint32\_t node\_image\_cnt){

for (uint32\_t node\_id = 2; node\_id <= node\_image\_cnt; node\_id++)

{

uint32\_t p = node\_id / 2 - 1;

auto&& p\_node = node\_image\_ptr\_vec[p];

uint8\_t committed\_version = !!(p\_node->flags &

(1 << (1 + (node\_id & 1) )) );

auto node\_image\_ptr = crack::read\_fs::read\_htree\_node\_image(fd,node\_id - 1,committed\_version);

node\_image\_ptr\_vec.emplace\_back(std::move(node\_image\_ptr));

}

}