The OpenSSH agent
Its use and custom hacks

Mirko Mariotti
Security course
MSc course in Computer Science
Department of Mathematics and Computer Science
University of Perugia

Monday 24th June, 2013

Abstract

In this paper I will explore the logic and the functionalities of the OpenSSH agent (*ssh-agent*) along with its counterpart *ssh-add*. After having shown its use and purposes, I will show some modification i made to the source code of openssh to improve such functionalities allowing the use of the agent as a general purpose system security "factotum".

Contents

1 Overview

1.1 SSH

SSH [1] (Secure Shell) is a network protocol that allows to establish a secure (encrypted) remote session on another host on a computer network, it is the "de facto" substitute of the Telnet. The SSH client has an interface and an usage similar to *telnet* and *rlogin* but the entire communication (both the authentication and the session) are encrypted instead of beeing "in clear" as its predecessor.

It is usually beeing used to:

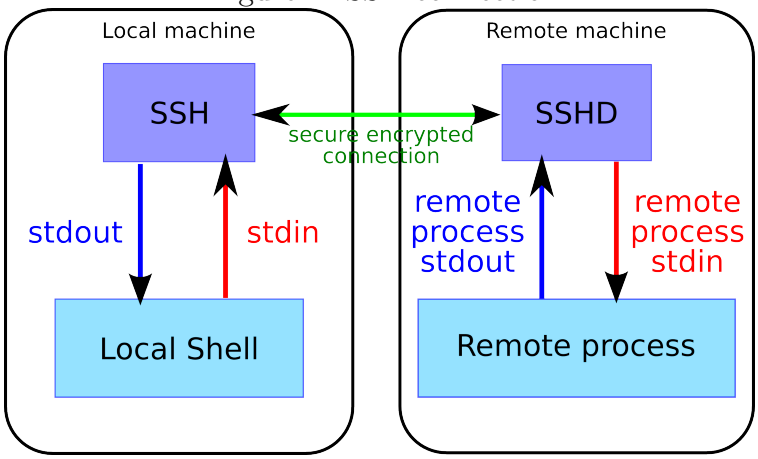
- Launch a shell (or in general spawn any command) on a remote machine in a secure way.
- Transfer files for and to a remote machine.

SSH software is avalaible for every kind of PC Operating Systems as well as on appliances, smartphones and tablets. The most used implementation of the protocol is *OpenSSH*, originally developed for the OpenBSD Operating System had been ported to many others systems. It is an open source software meaning that its source code is public and freely customizable. From now on every reference to SSH will mean to the OpenSSH implementation.

1.1.1 OpenSSH

SSH is made of a server software *sshd* and its client *ssh*. The server run on the remote machine and waits for connections usually on the TCP port 22. When invoked the client connects to the server and according to the authentication schema specified in the configuration it will asks for username and password or any other possible authentication information. Upon positive authtentication the client connects the remote file descriptors (via the server process) of the called process with itself letting the user think that a local process control a remote one as seen in Figure 1. With SSH is it possible also to transfer files from a remote machine and vice versa.

Figure 1: SSH connection



1.1.2 SSH usage

The SSH usage is straightforward:

- **Connect to a remote shell**:
  ssh username@remotehostname
- **Download files**:
  scp username@remotehostname:remotepath localpath
- **Upload files**:
  scp localpath username@remotehostname:remotepath

1.1.3 SSH authentication via keys

The username and password in the SSH authentication may be substituted by a key based authentication schema. Doing so make possible a password-less access to remote machines.The command *ssh-keygen -t dsa* allow the creation of a pair of DSA [3] key (public and private) as seen in the example in Figure 2. The private key is usually contained in the file *$HOME/.ssh/id_dsa* and may be protected or not by a passphrase asked by *ssh-keygen* at the moment of keys creation.
The public key usually is in the file *$HOME/.ssh/id_dsa.pub*.

```
$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/data/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/data/.ssh/id_dsa.
Your public key has been saved in /home/data/.ssh/id_dsa.pub.
The key fingerprint is:
05:58:76:c8:7b:d7:03:d4:3f:59:cb:2b:c9:3b:46:08 data@ambassador
The key's randomart image is:
+--[ DSA 1024]----+
|       ++.....   |
|      ..oo  . . .|
|         ..  o..+|
|        .E. . o=.|
|        S...o ..o|
|           . = . |
|            . o  |
|             +   |
|             . .  |
+-----------------+
```

Figure 2: DSA keys generation

By placing the public key within a file called *$HOME/.ssh/authorized_keys* on a remote machine (eventually on a different user home directory) we will instruct the *sshd* daemon on that machine to consider valid the connection originating from someone possessing the relative private key. If the private key is not protected by a passphrase the connection take place without any other requirements after issued the command. On the contrary if there is a passphrase the ssh command will ask for it prior to connecting. It may seem that protecting the key with a passphrase make unusefull the whole scenario since instead of a password we have to type a passphrase. It is not like that, a component of the SSH suite the *ssh-agent* allows to store keys and to let SSH use them whenever necessary. Like that the passphrase is need only once, when storing the key into the agent.

1.2 the OpenSSH agent

The OpenSSH agent is a daemon called *ssh-agent* that store and manages ssh identities. When an ssh command is issued, it search for the agent presence and if there is some key stored within. If is that the case and if such a key give access to a remote resource the ssh command execute and connects the user to the remote resource without asking for a password.

1.2.1 Agent usage

There are basically two ways of starting the agent:

- Upon starting the agent goes in background and write to stdout its environment parameters in the form of shell commands. so it can be launched from a shell this way: *eval 'ssh-agent'*
- The agent itself may spawn a command and export to it its own environment: *ssh-agent [command]*

When stating the agent several options may be specified via command line arguments:

- What kind of shell is the environment commands target.
- The default duration of a stored key: *ssh-agent -t [n]*, so after n seconds the key will expire and removed from the agent.
- The agent may be closed with: *ssh-agent -k*

1.2.2 Agent's client

ssh-add is the agent's client. The two of them communicate with a UNIX socket and ssh-add manage keys stored within the ssh-agent deamon via command line arguments. Several operation are possible:

- Add an identity (a pair of keys) to the agent: *ssh-add [filename]*
- Lock and unlock the agent: *ssh-add -x [-X]*
- List the fingerprints or the public key parameters of the managed identities: *ssh-add -l [-L]*
- Remove an identity or all: *ssh-add -d [filename] [-D]*

1.2.3 Sharing keys among machines

SSH provide a way to forward automatically keys from an endpoint to another using the option: *ForwardAgent=yes* of the ssh command. This option implicitly open an agent on the remote endpoint, propagate the keys on it and set the authentication environment to the called program. The remote program may now use the authentication information. In figure 3 is shown the process of a forwarding keys.

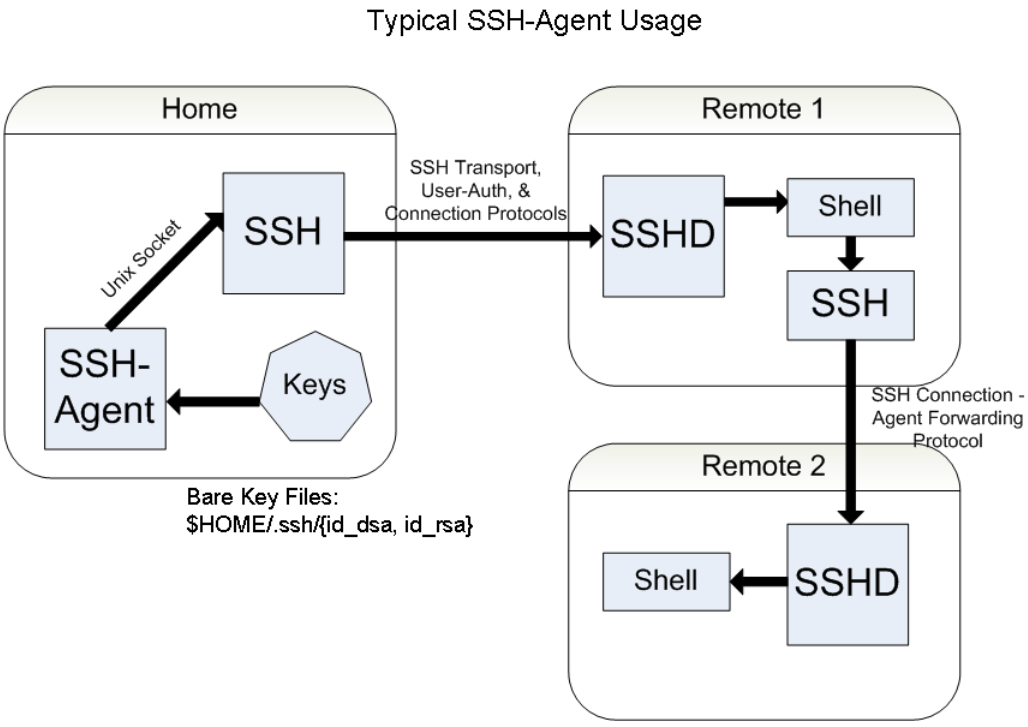


Figure 3: Agent forwarding

1.2.4 Some notes about security

As all ssh operations, an agent is not meant to be used in an insecure system. Indeed gain the access to the authentication socket opened by an agent gives an attacker access to all the systems protected by those keys. In the case of a forwarding agent among different system, on remote systems keys are never copied, but only endpoint to reach the main agent.

1.3 What is this work about

The agent is a powerfull tool that allows an easy and secure connection among machines via the ssh protocol, with it keys may be moved and used from a system to another but its use is specific to the ssh context. What if we want it to be a more general tool? If we want for example use a user's private key to create security tickets or to crypt and decrypt data with a symmetric algorithm. To do so, several modification is to be made to the openssh code. This modifications will be described in the subsequent sections.

2 OpenSSH agent internals

Before going any further it is necessary to take a look at the internal behaviour of the ssh-agent and the ssh-add. The involved source file are:

- *buffer.h*: It contains the functions to manage the buffers.
- *buffer.c*: It contains the functions implementations.
- *authfd.h*: It contains the functions definitions to interface with the "SSH authentication socket"
- *authfd.c*: It contains the functions implementations.
- *ssh-add.c*: It is the agent client source code.
- *ssh-agent.c*: It is the agent itself.

2.1 Interacting with the agent

When the agent starts it opens a UNIX socket that is the only interaction method possible, the socket location is generated by the agent as shell commands:

```
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-aOHLwBp25411/agent.25411; export SSH_AUTH_SOCK;
SSH_AGENT_PID=25412; export SSH_AGENT_PID;
echo Agent pid 25412;
```

Two environment variable *SSH_AUTH_SOCK* and *SSH_AGENT_PID* are generated. If a program of the OpenSSH suite find these variables it will try to talk with the agent via the socket. This is the "SSH authentication socket".

2.2 The buffer struct

All the communication done on the "SSH authentication socket" are managed on the source code via a struct build to manipulate fifo buffers that can grow if needed, It is called the "buffer struct". The "buffer struct" is defined in *buffer.h* as follows:

The buffer struct

```
typedef struct {
   u_char   *buf;    /* Buffer for data. */
   u_int  alloc;     /* Number of bytes allocated for data. */
   u_int  offset; /* Offset of first byte containing data. */
   u_int  end;    /* Offset of last byte containing data. */
}       Buffer;
```

Several functions are implemented in *buffer.c* and may be used to manipulate buffers such as create, free, add new data, etc. The agent and its client exchange buffers within the UNIX socket.

2.3 Agent Protocol

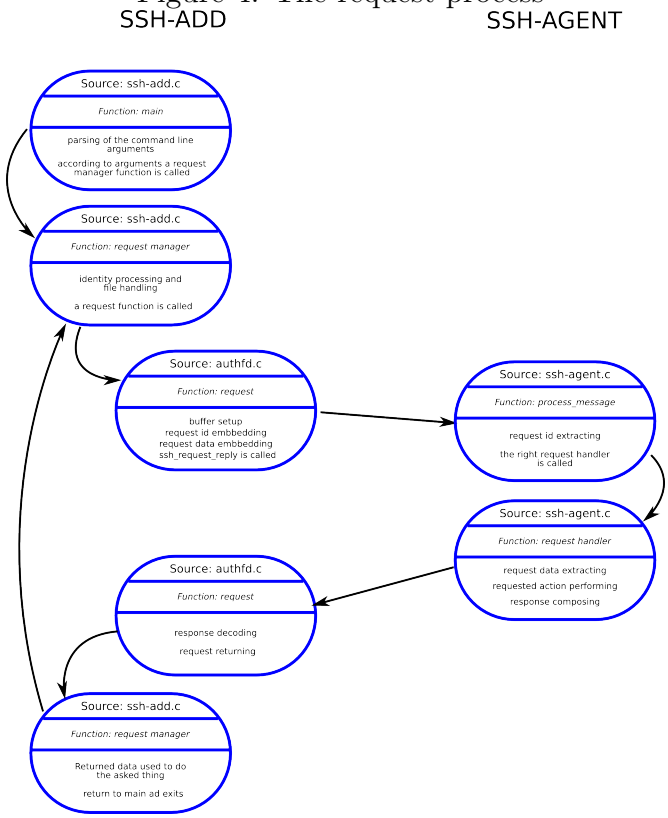
The atomic interaction beetween *ssh-add* and *ssh-agent* is in the form of a request-response query. The client compose e query starting with a byte that we may call *request ID* followed by all the data needed to process the request. The query is sent to the "SSH authentication socket" and the agent receive it and reading the *request ID* decide what to do. After having processed the request (for example the sing of a key) the agent compose a response starting with a *response ID* byte followed by the response data and send it back on the

socket. It also exists some generic failure and success responses. The protocol is described in the *PROTOCOL.agent* file while *authfd.h* has all the *request ID* and *response ID* in the form of C constants. The message maximum lenght is a constant so it may be necessary to have multiple request-response for a single *ssh-add* invocation. A function that we may call *request manager* can accomplish this task.

2.4 The Request Process

The figure 4 show the sequence of a typical request made from the *ssh-add* to the *ssh-agent*.

Figure 4: The request process



2.5 Data structures

Let's now check how the agent store the informations about its keys.

2.5.1 Identities

The agent store its managed keys in a linked list queue [2] of structs called Identity defined in *ssh-agent.c*. The struct definition is shown in listing 1.

Listing 1: The identity struct

```
typedef struct identity {
        TAILQ_ENTRY(identity) next;
        Key *key;
        char *comment;
        char *provider;
        u_int death;
        u_int confirm;
} Identity;
```

Each Identity entry stores:

- The effective keys, a pointer to the struct Key described in the next section.
- The identity lifetime, the duration in second of the key. After this amount o time the key will be automatically erased by the agent.
- A description.
- Some other optional parameter.

2.5.2 Keys

The key pointer in the Identity struct refer to a struct called Key and defined in *key.h* show in listing 2.

Listing 2: The Key struct

```
struct Key {
        int      type;
        int      flags;
        RSA     *rsa;
        DSA     *dsa;
        int      ecdsa_nid;     /* NID of curve */
#ifdef OPENSSL_HAS_ECC
        EC_KEY  *ecdsa;
#else
        void    *ecdsa;
#endif
        struct KeyCert *cert;
};
```

The functions needed to manage the keys kept here are functions that OpenSSH derive from the *openssl* library.

2.5.3 DSA Keys

For the purpose of this work we are interested only in DSA keys. The DSA keys are defined in the *openssl* library and in listing 3 is shown its definition.

Listing 3: The DSA Key struct

```
struct dsa_st
{
   int pad;
   long version;
   int write_params;
   BIGNUM *p;
   BIGNUM *q;      /* == 20 */
   BIGNUM *g;
   BIGNUM *pub_key;  /* y public key */
   BIGNUM *priv_key; /* x private key */

   BIGNUM *kinv;   /* Signing pre-calc */
   BIGNUM *r;      /* Signing pre-calc */

   int flags;

   // ...
};
```

In particular *priv_key* is the actual DSA private key, this is the key that will be used for the rest of this work.

3 Hacks

These are the improvement made:

- Added a ssh-add command option to get the amount of time (in seconds) that the key will stay on agent.
- Added a feature that allows to take a file and make an hash out of it after having merged it with the private key (create a sort of ticket).
- Added commands to crypt and decrypt files with a symmetric key derived from those stored in the agent.

3.1 Common traits

For every one of these the modification made are similar, expecially regarding the protocol extension, the add of command line arguments, the creation of new request functions and handlers.

3.1.1 Protocol

Every request need to have a unique request id and response id so it is necessary to define within *authfd.h* the new constants for these. The *process_message* function within *ssh-agent.c* has also to be modified to handle the new ids an point to the handlers functions as show in listing 4

Listing 4: process_message function in ssh-agent.c

```
        switch (type) {
// ...
        case SSH2_AGENTC_REQUEST_TIMELEFT:
                process_request_timeleft(e);
                break;
        case SSH2_AGENTC_REQUEST_CRYPT:
                process_request_crypt(e);
                break;
        case SSH2_AGENTC_REQUEST_DIGEST:
                process_request_digest(e);
                break;
// ...
```

3.1.2 Client Arguments

Every hack has to be invoked from *ssh-add* as command line arguments. That arguments has to be added to the source code of *ssh-add.c*. To do so the arguments has to be added to the main *getopt* call usign its conventions, the listing 5 show an example of such amodification. After the getopt opera-

Listing 5: Main getopt operations

```
while ((ch = getopt(argc, argv, "m:z:Z:TklLcdDxXe:s:t:o:")) != -1) {
   switch (ch) {
      ...
      case 'T':
         if (list_times(ac) == -1)
      ...
}}
```

tion each new argument has to be handled and the relative request manager function has to be called.

3.1.3 Request Manager

Every new functionality has its own manager function within *ssh-add.c*, the manager function prepare the data that will be sent to the real request (or to the real multiple requests if needed) and handle the responses. A function for every hack has to be included in *ssh-add.c* an example of one of these function is show in listing 6.

Listing 6: manager function example

```
static int make_digest(AuthenticationConnection *ac,char * datas) {
   //declarations
   //external file handling
   key = ssh_get_first_identity(ac, &comment, 2); // Key handling
   while (ret != 0) { pos=0;
      while ((ret != 0)&&(pos<MAX_AGENT_BUFFER)) {
         ret=read(fileinp,filedata+pos,MAX_AGENT_BUFFER-pos);
         if (ret == -1 ) { fprintf(stderr,"Failed"); return -1; } else { pos=pos+ret;
    } }
      if (pos !=0) {
         ecode = ssh_agent_digest(ac,key,&digest, &slen, filedata, pos); // real
    request
         for(i=0;i<slen;i++) printf("%02x",*(digest+i)); // Hack purpose
         free(digest); } }
   return ret; }
```

3.1.4 Request

Several new request submit functions have to be written in order to obtain new functionalities, these are declared in the file *authfd.h* and implemented in *authfd.c*. Each function is the atomic block of the agent-client comunication, it runs on the client and is responsible for the making the sigle request and getting back the response. Its standard sequence of actions are:

- Initialize a buffer.
- Write the request ID using the buffer functions
- Fill the buffer with the data needed to perform the requested operations.
- Send the request via a *ssh_request_reply* call.

- Check the response exit codes.
- Return with data from the response.

3.1.5 Request Handler

Each request has an handler function in *ssh-agent.c*. The handler receive the request buffer process it and compose the response buffer. The handlers are where the effective operation is done, for this reason they are part of the agent's code. For example the operation of crypt a block of code is done by an handler function. The handler is called by the agent every time a request arrives as it is the atomic response operation.

3.2 Timeleft hack

As seen the key struct within the agent store in the deathtime field how many seconds the key will stay in the agent before beeing removed. The standard openssh implemetation does not have a way to get this value. The timeleft hack add the *-T* option to ssh-add allowing the user to get this information. This value may be put in a desktop widget, or in a command line prompt.

3.2.1 Examples

Agent with no key

```
$ eval `ssh-agent` ; ssh-add -l ; ssh-add -T
The agent has no identities.
none
```

Agent with an expiring key

```
$ eval `ssh-agent` ; ssh-add -t 7200 ; ssh-add -l ; ssh-add -T
[passphrase ask ...]
1024 ea:98:a2:1d:f7:18:a1:11:22:2e:14:39:1b:66:63:3b /home/mirko/.ssh/id_dsa (DSA)
7198 sec
```

Agent with a not expiring key

```
$ eval `ssh-agent` ; ssh-add ; ssh-add -l ; ssh-add -T
[passphrase ask ...]
1024 ea:98:a2:1d:f7:18:a1:11:22:2e:14:39:1b:66:63:3b /home/mirko/.ssh/id_dsa (DSA)
forever
```

3.2.2 Request and Handler

Since the task of retriving the time left is very simple (only one request) the request manager is trivial. It purpose is just to write the duration. “forever” or “none”. The request (listing 7) is only the request ID since there is no other argument. The replay sends back an integer as show in listing 8.

Listing 7: Timeleft Request

```
int
ssh_get_identity_timeleft( AuthenticationConnection *auth)
{
   int howmany;
   Buffer msg;
   buffer_init(&msg);
   buffer_put_char(&msg, SSH2_AGENTC_REQUEST_TIMELEFT);
   if (ssh_request_reply(auth, &msg, &msg) == 0) {
     buffer_free(&&msg);
     return 0;
   }
   type = buffer_get_char(&msg);
   if (agent_failed(type)) { return 0; }
   else if (type != SSH2_AGENT_TIMELEFT_ANSWER) {  return 0; }
   howmany = buffer_get_int(&msg);
   buffer_free(&msg);
   return howmany;
}
```

Listing 8: Timeleft Handler

```
static void
process_request_timeleft(SocketEntry *e, int version)
{
  Idtab *tab = idtab_lookup(version);
  //...
  buffer_init(&msg);
  buffer_put_char(&msg,
    SSH2_AGENT_TIMELEFT_ANSWER);
  id = TAILQ_FIRST( &tab->idlist);
  //...
  buffer_put_int(&msg, id->death);
  //...
}
```

3.2.3 Request Manager

In listing 9 is shown the code for the Timeleft Request Manager

Listing 9: Timeleft Request Manager

```
static int
list_times(AuthenticationConnection *ac)
{
  int deathtime;
  int currenttime;
  deathtime=ssh_get_identity_timeleft(ac);
  currenttime=time(NULL);

  if (deathtime == 0 )
  {
    fprintf(stdout,"forever\n");
  }
  else
  {
    if ( deathtime > currenttime )
    {
      fprintf(stdout,"%d sec\n",
        deathtime-currenttime);
    }
    else
    {
      fprintf(stdout,"none\n");
    }
  }

  return 0;
}
```

3.3 Token hack

With the Token hack is possible to create an hash of a file merged with the agent's private key. The hack add the *-m [file]* option to ssh-add allowing the user to specify the file (on stdin with -). To compute the token this procedure has been choosen:

- The file (or stream) to be used is splitted in chunks smaller the maximum message length between *ssh-agent* and *ssh-add*.
- For every chunk:
  - A request is made
  - The handler concatenate the private key with the chunk and make an SHA1 hash
  - The hash is sent back
- The request manager concatenate into the final result all the hashes.

This value may be use as token to authenticate other services.

3.3.1 Examples

Token from a file

```
$ echo "This is a test" > testfile ; ssh-add -m testfile
d2aa41840bebdf183ee5bc4b4104716dc3342f7b
```

Token from stdin

```
$ echo "This is a test" | ssh-add -m -
d2aa41840bebdf183ee5bc4b4104716dc3342f7b
```

3.3.2 Request Manager

The Token hack request manager does the following actions:

- Open the file source for the token.
- Spilt it in several parts and for every part:
  - Compose a token request
  - Send the request
  - Print the result request
- Use the concatenation of the partial results as final token

In listing 10 parts of the Token request manager are shown.

Listing 10: Token Request Manager

```
static int make_digest(AuthenticationConnection *ac,char * datas) {
   // Variables omitted
        if (datas!=NULL) {
                fileinp=open(datas,O_RDONLY);
                if (fileinp == -1) return -1;
                lseek(fileinp, 0, SEEK_SET);
        } else { fileinp=fileno(stdin); }
        key = ssh_get_first_identity(ac, &comment, 2);
        ret=-1;
        while (ret != 0) {
                pos=0;
                while ((ret != 0)&&(pos<MAX_AGENT_BUFFER)) {
                        ret=read(fileinp,filedata+pos,MAX_AGENT_BUFFER-pos);
                        if (ret == -1 ) {
                                fprintf(stderr,"File read failed"); return -1;
                        } else { pos=pos+ret; }
                }
                if (pos !=0) {
                        ecode = ssh_agent_digest(ac,key,&digest, &slen, filedata, pos);
                        if (ecode != 0 ) { fprintf(stderr,"Digest failed"); return -1;
    }
                        for(i=0;i<slen;i++) printf("%02x",*(digest+i));
```

```
                    free(digest);
                }
        }
    // ...
        return ret;
}
```

3.3.3 Request and Handler

The request function (shown in listing 11) compose the buffer like this:

- Buffer initializzation
- Request ID write
- Request Data write
- Request Submit
- Return the result to the calling function

Listing 11: Token Hack Request

```
int ssh_agent_digest(AuthenticationConnection *auth, Key *key, u_char **sigp ...) {

   // ...

   buffer_init(&msg);
   buffer_put_char(&msg, SSH2_AGENTC_REQUEST_DIGEST);

   // ...

   buffer_put_string(&msg, blob, blen);
   buffer_put_string(&msg, data, datalen);
   buffer_put_int(&msg, flags);

   // ...

   if (ssh_request_reply(auth, &msg, &msg) == 0) { buffer_free(&msg); return -1; }
   type = buffer_get_char(&msg);
   if (agent_failed(type)) {
      logit("Agent admitted failure to sign using the key.");
   } else if (type != SSH2_AGENT_DIGEST_ANSWER) {
      fatal("Bad authentication response: %d", type);
   } else {
      ret = 0;
      *sigp = buffer_get_string(&msg, lenp);
   }

   buffer_free(&msg);
   return ret;
}
```

Handler shown in listing 12 respond the request queries performing the following operations:

- Get the key specs and data
- Get the the private key
- Concatenate the private key with the data and make a SHA digest
- Send back the result

Listing 12: Token Hack Request Handler

```
static void process_request_digest(SocketEntry *e) {

   // ...

   blob = buffer_get_string(&e->request, &blen);
   data = buffer_get_string(&e->request, &dlen);

   flags = buffer_get_int(&e->request);

   // ...

   key = key_from_blob(blob, blen);

   // ...

   ppkk=BN_bn2hex(id->key->dsa->priv_key);

   // ...

   intre=(unsigned char *) malloc ((strlen(ppkk)+dlen+1)*sizeof(unsigned char));
   memset(intre,0x00,strlen(ppkk)+dlen);
   memcpy(intre,ppkk,strlen(ppkk));
   memcpy(intre+strlen(ppkk),data,dlen);
   SHA((unsigned char *) intre, strlen(ppkk)+dlen, (unsigned char *) &digest);

   // ...

   buffer_init(&msg);
   buffer_put_char(&msg, SSH2_AGENT_DIGEST_ANSWER);

   // ...

   buffer_put_string(&msg, digest, SHA_DIGEST_LENGTH);

   // ...

}
```

3.4 Crypt/Decrypt hack

The Crypt/Decrypt hack adds the possibility to encrypt and dectrypt files using the private key (or an hash from it derived) as symmetric key. The hack add the *-z [file]* option to encrypt a cleartext file to the ciphertext specified

with the *-o [file]* option, and the *-Z [file]* to do the opposite (with the *-o [file]* option as well). Internally the AES symmetric cipher is used.

3.4.1 Examples

Crypt and Decrypt

```
$ echo "This is another test" > plaintext
$ hexdump plaintext
0000000 6854 7369 6920 2073 6e61 746f 6568 2072
0000010 6574 7473 000a
0000015
$ ssh-add -z plaintext -o ciphertext
$ hexdump ciphertext
0000000 0000 2000 4e56 27c1 47c1 812a 5205 1f86
0000010 cbd3 f152 b783 6c13 c505 e42v b7ce a809
0000020 58fd 0757
0000024
$ ssh-add -Z ciphertext -o plaintext_new
$ hexdump plaintext_new
0000000 6854 7369 6920 2073 6e61 746f 6568 2072
0000010 6574 7473 000a
0000015
```

3.4.2 Request

The request is almost identical to the one from the Token hack and shown in listing 13.

Listing 13: Crypt/Decrpy request

```
int ssh_agent_crypt(AuthenticationConnection *auth, Key *key, u_char **sigp, ...) {
   // ..
   if (en_or_de != 0) en_or_de=1;

   buffer_init(&msg);
   buffer_put_char(&msg, SSH2_AGENTC_REQUEST_CRYPT);
   buffer_put_string(&msg, blob, blen);
   buffer_put_string(&msg, data, datalen);
   buffer_put_int(&msg, en_or_de);
   buffer_put_int(&msg, flags);
   xfree(blob);

   if (ssh_request_reply(auth, &msg, &msg) == 0) {
      buffer_free(&msg); return -1;
   }
   type = buffer_get_char(&msg);
   if (agent_failed(type)) { logit("Agent admitted failure to sign using the key.");
   } else if (type != SSH2_AGENT_CRYPT_ANSWER) { fatal("Bad authentication response:
    %d", type);
   } else { ret = 0;
      *sigp = buffer_get_string(&msg, lenp);
   }
   buffer_free(&msg);
   return ret;
}
```

3.4.3 Handler

To handle the AES cryptography i used the *openssl EVP library* that provides high-level interface to cryptographic functions, and created 3 functions within *ssh-agent.c*:

Listing 14: AES Initialization

```
int aes_init(unsigned char *key_data, int key_data_len, unsigned char *salt, ... ) {
  // ...

  EVP_CIPHER_CTX_init(e_ctx);
  EVP_EncryptInit_ex(e_ctx, EVP_aes_256_cbc(), NULL, key, iv);
  EVP_CIPHER_CTX_init(d_ctx);
  EVP_DecryptInit_ex(d_ctx, EVP_aes_256_cbc(), NULL, key, iv);

  return 0;
}
```

Listing 15: AES Crypt

```
unsigned char *aes_encrypt(EVP_CIPHER_CTX *e, unsigned char *plaintext, int *len)
{
  int c_len = *len + AES_BLOCK_SIZE, f_len = 0;
  unsigned char *ciphertext = malloc(c_len);

  EVP_EncryptInit_ex(e, NULL, NULL, NULL, NULL);
  EVP_EncryptUpdate(e, ciphertext, &c_len, plaintext, *len);
  EVP_EncryptFinal_ex(e, ciphertext+c_len, &f_len);
  *len = c_len + f_len;
  return ciphertext;
}
```

Listing 16: AES Decrypt

```
unsigned char *aes_decrypt(EVP_CIPHER_CTX *e, unsigned char *ciphertext, int *len)
{
  int p_len = *len, f_len = 0;
  unsigned char *plaintext = malloc(p_len + AES_BLOCK_SIZE);

  EVP_DecryptInit_ex(e, NULL, NULL, NULL, NULL);
  EVP_DecryptUpdate(e, plaintext, &p_len, ciphertext, *len);
  EVP_DecryptFinal_ex(e, plaintext+p_len, &f_len);

  *len = p_len + f_len;
  return plaintext;
}
```

The functions are then used in the request handler:

Listing 17: Crypt/Decrypt handler

```
static void process_request_crypt(SocketEntry *e) {
// ...
   EVP_CIPHER_CTX en, de;
   unsigned int salt[] = {12345, 54321};
// ...
   ppkk=BN_bn2hex(id->key->dsa->priv_key);
// ...
   buffer_init(&msg);
```

```
    buffer_put_char(&msg, SSH2_AGENT_CRYPT_ANSWER);
    if (aes_init(ppkk, strlen(ppkk), (unsigned char *)&salt, &en, &de)) {
// ...
    if (en_or_de==0) {
        crypttext = aes_encrypt(&en, (unsigned char *)data, &len);
    } else {
        crypttext = (unsigned char *)aes_decrypt(&de, (unsigned char *)data, &len);
    }
    buffer_put_string(&msg, crypttext, len);
// ...
    EVP_CIPHER_CTX_cleanup(&en);
    EVP_CIPHER_CTX_cleanup(&de);
// ...
}
```

3.4.4 Request Manager

The request manager (called crypt_thigs) is much complicated than the others in many ways, this is a list of the problems and solutions found:

- The output of the process has to be stored in a file so it have been necessary to include one more command line arguments (*-o*)to *ssh-add.c* to specify the file name for the output. Furthermore this file has to written within the request manager.
- It has to handle both encryption and decryption so has been added a flag to the signature of the crypt_thing function to specify if it is crypt or decrypt.
- The lenght of AES ciphertext may be different (and in general is) from the plaintext lenght. This means that a minimal structure is needed for the output file, keeping the length of each block just before the ciphertext.
- The endianness-neutrality and data type length of the file write operations have to be assured to have portability. The use of C99 standards for data type (as uint32_t) solve the data type length while i used the network ordering function to have endianness-neutrality (as htonl).

3.5 Regression tests

The openssh sources has a Makefile target that provides a suite of regression tests for all the protocols and keys operations among different versions and architectures. I added tests to check the determinism and coherence of the

tokens and ciphertexts, in table 1 are the systems where the code has been tested succesfully.

Table 1: Tested Systems

Architecture	Operating System	Kernel
amd64	Gentoo	Linux 3.8.13
i386	OpenBSD	OpenBSD 5.0 GENERIC
amd64	Debian Squeeze	Linux 2.6.32
amd64	Debian Wheezy	Linux 3.2.0
i686	Debian Wheezy	Linux 3.2.0
alpha EV67 Tsunami	Gentoo	Linux 3.7.10

4 Use cases

Now let's see some examples that may benefit from this improved *ssh-agent*.

4.1 Using the token to unlock Luks devices

Linux Unified Key Setup or LUKS is a disk-encryption specification, it allows a user to encrypt a device with one ore more passphrase:

Creating a LUKS device

```
cryptsetup luksFormat /dev/sdp [keyfile]
```

Add a key to a LUKS device

```
cryptsetup luksAddKey /dev/sdp [keyfile]
```

Decrypt a LUKS device

```
cryptsetup luksOpen /dev/sdp cryptodevicename [--key-file keyfile]
```

As seen from the examples it is possible to use a keyfile.

Following this procedure is then possbile to bond a LUKS device with a SSH private key using the Token hack:

- Create a random file used as plaintext file:
  dd if=/dev/urandom of=plaintext bs=1 count=1024
- Start the agent and store an identity: *eval 'ssh-agent' ; ssh-add*
- Create a token: *ssh-add -m plaintext > token*
- Use the token to format the device or add it to an existing one:
  cryptsetup luksFormat /dev/devn token
  or *cryptsetup luksAddKey /dev/devn token*
- Delete the token.
- From now on the device may be unlocked recreating the token and with: *cryptsetup luksOpen /dev/devn crytodevn –with-keyfile token*

4.2 HIDS

4.2.1 Tripwire

Tripwire is a free software to monitor the data integrity on systems. Acording to a configurable policy it keeps a cryptographic database of data and alert for changes against the policy. It is an HIDS (host-based intrusion detection system).
It works with two passphrases called "local" and "site". The "site" passphrase is meant to be the main one, used to modify the policy and the configuration and work site-wide. The "local" is used to update the cryptographic database.
Tripwire is a sofisticated system and need continuos adjustments so bonding the passphrases to the SSH keys may simplify the operations. Moreover with this method the "site" passphrase may travel accross servers using the forwarding capabilities of SSH

4.2.2 Implement a remote HIDS

The following scripts show how to implement a remote HIDS with the improved agent:

Creating the database

```
#!/bin/bash
CONFLIST=""
CONFLIST="$CONFLIST /bin/ps"
CONFLIST="$CONFLIST /usr/bin/ssh-agent"
DBLOC=/tmp/dbloc

for i in $CONFLIST; do
  REMOTE_TOKEN=`ssh -o ForwardAgent=yes myremotehost "ssh-add -m $i"`
  echo "$i -> $REMOTE_TOKEN"
  touch $DBLOC/$REMOTE_TOKEN
done
```

Check the remote system

```
#!/bin/bash
CONFLIST=""
CONFLIST="$CONFLIST /bin/ps"
CONFLIST="$CONFLIST /usr/bin/ssh-agent"
DBLOC=/tmp/dbloc

for i in $CONFLIST; do
  REMOTE_TOKEN=`ssh -o ForwardAgent=yes myremotehost "ssh-add -m $i"`
  REMOTE_TOKEN2=`ssh myremotehost "cat $i" | ssh-add -m -`
  if [ -f $DBLOC/$REMOTE_TOKEN ]; then
    echo "$REMOTE_TOKEN -> $i (ok remote)"
  else
    echo "Alert !"
  fi
  if [ -f $DBLOC/$REMOTE_TOKEN2 ]; then
    echo "$REMOTE_TOKEN2 -> $i (ok local)"
  else
    echo "Alert !"
  fi
done
```

5 Conclusion

In this work i tried to give to the openssh agent a central role in my systems security and to make easier the every day security checks. The agent is now capable of generating tokens that may be used to control other systems and to crypt and decrypt file using the stored keys. The whole system works well and have been longed tested, some work has to be done to put the sources in an elegant way (some docs, comments, check for error states, etc). Some further improvments may be done such as:

- Let the user choose the digest algorithm and not only use the SHA.
- Let the user choose symmetric cryptography algorithm other than AES.
- Improve the regression tests introducing more samples.

- Introduce a random sequence whose token is used as symmetric key instead of using the key directly (It will need a protocol enhancement since it will be necessary to store the random sequence in the ciphertext).
- Manage multiple keys and other then DSA.

References

[1] D. J. Barrett and R. E. Silverman. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[2] OpenBSD. Openbsd programmer's manual, 2011. [Man page].

[3] Wikipedia. Digital signature algorithm — Wikipedia, the free encyclopedia, 2013. [Online; accessed 16-June-2013].