Filesystem encoding by the use of USB-Sticks

SYSTEMENTWICKLUNGSPROJEKT

Development:
Johann Hacker

Supervisor:
Dr. Helmut Reiser, Harald Roelle

Assignment:
Prof. Dr. H.-G. Hegering

Table of Contents

I. Introduction

We live increasingly in an information society. Computers are no longer only used for typing documents or spread-sheet calculations. To reduce expenses in enterprises and institutions (e.g. universities) computers are interconnected. In this kind whole departments can share various resources (e.g. printers, data, etc..). Over so-called 'Groupware' the work can be settled more efficiently. Almost any computer today is permanently or at least occasionally connected with the Internet. In order to write e-mail, to look for information or to exchange data. Larger companies and institutions interconnect their settlements over the Internet. By this global cross-linking over partly to completely public networks, as for instance the Internet, one has the problem that data can be read by everyone. For the protection of sensitive data cryptography offers a way out. To guarantee safe communication between the single settlements of companies VPN (virtual private networks) can be used. It acts over strongly encoded connections, which use the Internet as carrier-net. The data within the networks at each settlement is however still readable for each internal user. Around to prevent that, one can give every user different rights of access and passwords. This however, only protects the data as long as the users have no physical access to the hardware. Because in this case a user could simply reboot the computer with a boot disk or install the storage medium into another computer to simply give himself the necessary rights. Here one can also fall back again to cryptography and simply encrypt the sensitive data. Single files can be encrypted without larger expenditure. In UNIX environments one can use e.g. gpg (GNU Privacy Guard). To encrypt a vast amount of files this procedure is unmanageable. In this case, it is more comfortable to encode whole file systems.

II.Setting of task

The task of this 'Sytementwicklungsprojekt' (SEP) is it to develop a program which transparently allows one or more users to work with encrypted file systems. The users should authenticate them selfs with common USB sticks. The users work on pure X-terminals. That means, on these computers only a Linux Kernel and a X-server runs. All programs and data are stored on a file server and are also executed on it (X-clients).
If a user wants to access his encrypted files, he only needs to plug his USB stick into a X-terminal. From now on the encrypted data can be accessed. If the USB stick is unplugged again no further access is possible.
To keep the program, to be developed in this SEP, easy (and/or more easily) to maintain, as less own code as possible should be provided. Instead mainly existing software (libraries and programs) is to be used.

III.Declarations

To assist in the understanding of the following text I will explain some important terms.

Block device
A block device is an I/O device which stores it's information in fixed-size blocks, each one with it's own address. Common block sizes range from 512bytes to 32768bytes. Block devices are: floppy drives, hard drives, CDROM-drives, USB-Sticks, ...

Loopback device
The loopback device allows to use a regular file as a block device. You can then create a file system on that block device and mount it just as you would mount other block devices. Additionally a whole block device can be accessed by a loopback device. This only makes sense if the data passed from the normal block device to the user gets altered by the loopback device. E.g. the cryptoloop mechanism where the data passed through gets de- or encrypted.

The proc filesystem
The proc filesystem is a characteristic of Linux. It makes information available about the current condition of the Linux Kernel as well as about the current processes. There are files and directories for process-independent information like loaded modules, used bus-systems, etc.. Additionally it permits in a simple manner the change of some kernel parameters at run-time.

base64 data representation format
The base64 data representation format doesn't encrypt or compress data. It converts data into a 64-character alphabet. Each of these 64 characters is printable.
Binary files contain non-printable characters. By converting binary files into base64 these binary files become printable (e.g. can be printed to stdout).
The base64 is described in detail in the RFC 3548.

IV.Cryptography - a short overview

AES – Cipher

NIST (National Institute of Standards and Technology), decided that the US. Federal Government needed a new cryptographic standard (for the former DES – Data Encryption Standard). So they announced a cryptographic contest where researchers all over the world were invited to submit proposals for the new standard, to be called AES (Advanced Encryption Standard).
The winner of this contest was the Rijndael algorithm (named after its inventors, Joan Daemen and Vincent Rijmen).
The Rijndael algorithm is a symmetric block cipher which supports key lengths of 128, 192 and 256 bits. Symmetric means the use of the same key for de- and encoding. And block cipher means that an n-bit block of plain data will be transformed into an n-bit block of encoded data. For more information about the Rindael cipher read the documentation on the official homepage [1].
This cipher is quite fast and it's design has been made public. Therefore many free implementations exist. These are the reasons why this cipher is used in the later described programs.

Secure keys

As mentioned above, AES uses keys to encrypt data. Therefore a few thoughts about the right choice concerning the key selection have to be made. The following information was taken from a document which can be found at [2].

Why keys can be insecure
Why keys can be insecure, can have several reasons. The simplest e.g. would be, that it is too short: a 2 character long key has (if a-z, A-Z and 0-9 are chosen as valid characters) 61^2 (= 3721) possible combinations. Surely, for a human that is very much to try all possible combinations, but a PC would only need a few seconds.
A further reason is the structure of a key. The choice for e.g. '111' as key wouldn't be very wise. On the one hand it is very easy to find out (squint on the keyboard) and in addition the hash generated from it is not very complex.
A very bad choice would be the own bank-account or telephone number as key. A potential attacker would try these first. The reason is that most people are unwilling to remember additional unfamiliar cryptic keys.
The three most popular words are 'god', 'sex' and 'love'. Likewise popular are parts of the e-mail address and/or the name, the user ID, etc..
Additionally the shouldn't be a word in any known language.

How long should a key be?
The question is not to be answered so easily. To the estimate take a look at figure 1 (possible key characters are a-z,A-Z,0-9 = 61; brute-force attack) :

Key length	**required time (assumption is one million characters per second)**
3 characters	~ 0,2 seconds
5 characters	~ 14 minutes
8 characters	~ 53252 hours
10 characters	~ 1179469 weeks
12 characters	~ 84168853 years
15 characters	~ 19104730610573 years

Figure 1

All these data are **maximum times**! Maximum time means: if someone tries all possible key combinations and he succeeds with the last possible permutation.
Additionally there are various possibilities (dependent on the used cipher) to calculate the key (e.g. from the hash value) and thus be faster successful.

Humans tend to choose far to short and/or too insecure passwords. To solve this problem one can use a keygenerator.

Chosen Key generator
As key generator the Perl program secpass [3] of Sam Trenholme was modified. The chosen key length is 32 characters (32 * 8 bit = 256 bit) the maximum key length of the AES-cipher.
For random number generation the '/dev/urandom' device which the linux kernel offers was used. This device produces quite good random numbers which are read byte by byte. The algorithm works like this:

A character set is given as an array. 64 different characters are permitted, 'A..N', 'P..Z', 'a..k', 'm..z', '2..9', '@', '.', '!', '+', '-', '*'. Because 1 ~ = l and 0 ~ = O look the same with various fonts Sam Trenholme left these out. Although the keys generated here have never to be entered by a human, this wasn't changed.

1. read one byte from /dev/urandom
2. byte = byte & 63 (=> only values between 0 and 63 are left)
3. lookup array[byte] (=> one character from the set) => first character of the key
4. continue this until the desired key length is reached

Linux and Cryptography - Encoded file systems

Linux offers the possibility to encode complete file systems with the help of the loopback device. One avails oneself of the cryptographic functions of the core (Crypto API).

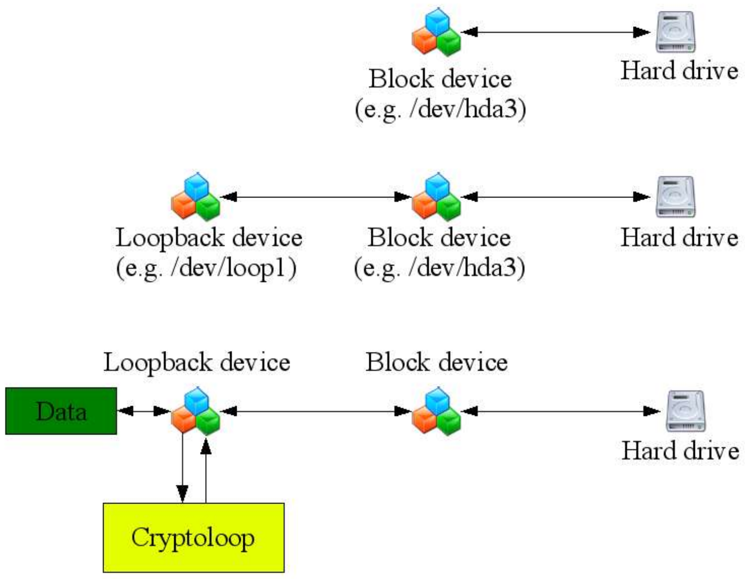

Figure 2

The loop devices work as intermediate layer between a block device (e.g. /dev/hda3) and the filesystem contained on it. Under normal conditions the filesystem directly accesses the block device (figure 2, above). The loopback device behaves like a block device, with the exception that it gives the data to a normal device or to a file (figure 2, center). This procedure is known to everyone who at least once has mounted an ISO image with the 'loop' option.
Loopback devices are also able to change the data passing through. With cryptoloop the kernel intervenes in each writing and reading, in order to de- or encode the data (figure 2, down).
The user notices -except to the initial password inquiry- nothing. One task of this SEP is to hide this password inquiry from the user by the aid of USB sticks.

V.Preparing Linux and the Kernel

Thus Linux can deal with the USB-Bus, USB-Storage devices and cryptography, the following kernel options have to be activated.
(Y|M means: **Y**es gets compiled into the kernel or will be built as **M**odule; Y means: Compile into the kernel)

- Loopback device support [CONFIG_BLK_DEV_LOOP] = Y|M
- SCSI Support [CONFIG_SCSI] = Y|M
- SCSI disk support [CONFIG_BLK_DEV_SD] = Y|M
- /proc file system support [CONFIG_PROC_FS] = Y
- Support for USB [CONFIG_USB] = Y|M
- Preliminary USB device filesystem [CONFIG_USB_DEVICEFS] = Y

Choose the appropriate driver for your USB-Host controller:
- EHCI HCD (USB 2.0) support [CONFIG_USB_EHCI_HCD] = Y|M
- UHCI (Intel PIIX4, VIA, ...) support [CONFIG_USB_UHCI] = Y|M
- UHCI Alternate Driver (JE) support [CONFIG_USB_UHCI_ALT] = Y|M
- OHCI (Conpaq, iMacs, OPTi, SiS, Ali, ...) support [CONFIG_USB_OHCI] = Y|M

– USB Mass Storage support [CONFIG_USB_STORAGE] = Y|M

In order to be able to use encoded file systems, the following kernel options have to be activated too:

- Cryptographic API [CONFIG_CRYPTO] = Y

Additionally the en-/decryption algorithms must be selected which one wants to use later.

In order to be able to use encryption with the loopback devices, the kernel has to be patched. One must download the cryptoloop Patch [4]. Now one must unpack the patch and follow the instructions in the README.
How to patch the tools 'mount' and 'losetup' from the util-linux packet also is described in this README.
If one uses a newer (since 8.0) SuSE distribution with the default kernel and the SuSE util-linux package, all adjustments are already made. With the Debian distribution all adjustments, except the cryptoloop patch, are already made.

Thus proc and usbdev file systems are available the following lines should be supplied in /etc/fstab:
```
proc        /proc          proc      defaults 0 0
usbdevfs    /proc/bus/usb  usbdevfs  defaults 0 0
```

VI.Possible Solutions

As already explained in section II, a Client/Server environment must be provided. Since secret data has to be transmitted (e.g. passwords), one must provide a secure communication between the x-terminals and the server. The USB sticks are plugged-in at the x-terminals. One must ensure that the USB stick is correctly recognized and mounted/unmounted. Additionally the data on the USB stick must be examined for crypto information and submitted to the server. For the server a program must be developed which evaluates the data conveyed by the x-terminal and takes care for the mounting/unmounting of the encrypted filesystems.

To solve the task it is best to divide it into three parts:
1. Secure data communication between X-terminals and server
2. Observation of the USB bus and recognition of USB sticks by the X-terminals
3. Process the data sent from the X-terminal about mounting/unmounting of the encrypted file systems

Next two possible solutions are presented followed by a comparison of their advantages and disadvantages.

One Server Instance

For the transmission of the data OpenSSH is used. OpenSSH offers the possibility to establish a secure encoded communication over an insecure network. It was particularly interesting because OpenSSH also offers the possibility to execute a program on a remote host instead of a Shell. The standard-input of the remote host is connected the standard-output of the local terminal and vice versa. By that way one can transmit data relatively easy and secure between two computers.

On the X-terminal a program (mount_server) runs permanently with root-rights, which supervises the USB bus. When a USB stick is recognized it gets mounted and is examined for crypto information. If crypto information is found, it gets readout. As mentioned above an OpenSSH connection is established. By OpenSSH a program (mount_client) is executed on the Server and the determined data is transfered there.

This 'mount_client' builds up a Socket connection to a program (crypt_server) which is permanently running on the server and forwards the incoming data of the X-terminal.
'crypt_server' is a real server application. I.e. it listens on a port for new 'mount_client's with the desire to connect. Additionally it supervises all ports to which clients already have connected for new available data. If one client announces a new USB stick with crypto information, the encoded filesystem associated is mounted. With a message of an unplug event the encoded filesystem gets unmounted immediately.

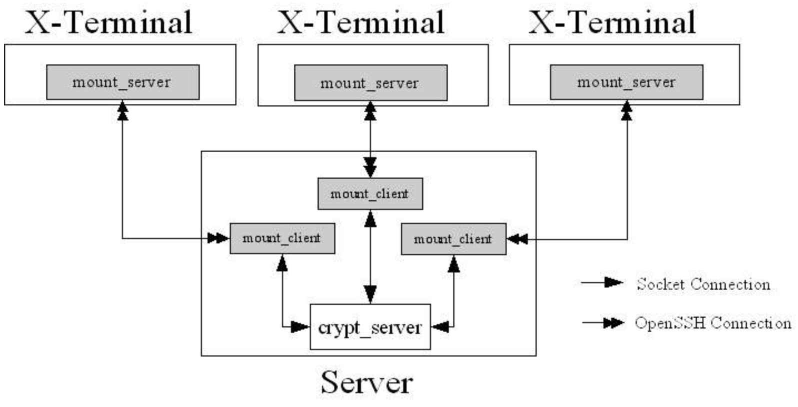

Figure 3 – Solution One

For safety reasons, the file systems in use of a connected X-terminal are unmounted with the loss of the connection between X-terminal and server and/or between 'mount_client' and 'crypt_server' immediately. Figure 3 (above) shows solution one.

Multiple Server Instaces

When logging in, the login data is rerouted by the X-terminal to the server. One can latch oneself in the login process at the server and start the program 'usb_crypt_manager.pl' (with root rights; cf. Crypto Manager – fig. 4).
As described in the first solution the decision fell on OpenSSH in the second solution too, to establish a secure connection.
This 'manager' starts thus with OpenSSH a shell script (usb_scan.sh; cf. USB Watch – fig. 4) on the X-terminal. This shell script supervises the USB bus. If a USB stick is recognized, it is mounted and examined for crypto information. If a USB stick contains crypto data it immediately is conveyed over the OpenSSH connection to the 'usb_crypt_manager.pl'. Pulling a USB stick is likewise conveyed to the server.
The 'usb_crypt_manager.pl' coordinates the mounting and unmounting of the encoded file systems. If the OpenSSH connection for any reason should tear off, the mounted crypto file systems for safety reasons are unmounted immediately. Figure 4 (down) shows solution two.

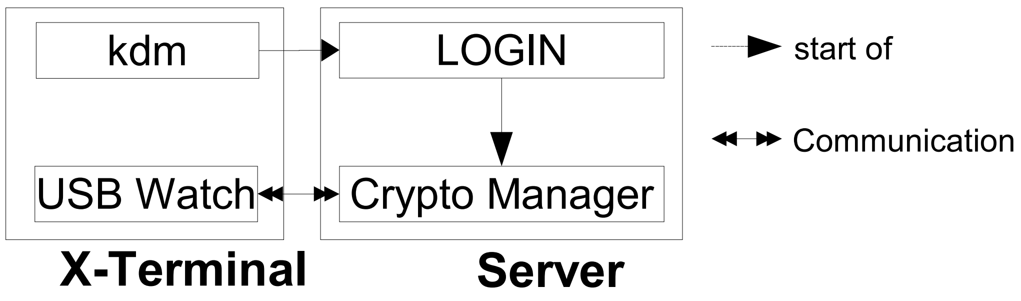

Figure 4 – Solution Two

Chosen Solution - Multiple Server Instaces

Compared with solution one, solution two has a much easier way of communication. It has only one channel where data is passed on. Solution one additionally needs one program ('mount_client') more as relay. Therefore in solution two less error-handling has to be done.
Negative in solution one is the 'big' server application which has to deal with a lot of connected clients simultaneous. In contrast to solution two where there is a 'small' manager application that only has to deal with one X-terminal.
Therefore the winner of this comparison is solution two. The next section describes the implementation of solution two.

VII.Implementation

Used data/file structures

File Structure on the USB stick
For every encrypted container file a file is located on the USB stick in the directory '.crypt'. It contains two lines. In line one is the user name of the user owning the stick followed by a semicolon and a newline. Line Two contains a AES encrypted key. The key is stored base64-encoded on the stick because that way the file can be printed to stdout without loss.

So the file (without base64 encoding) looks like this:

```
<user>;\n
<password>
```

Structure of the server configuration file
The configuration file consists of individual lines. Each line contains the information needed to mount a encrypted filesystem for a user. The elements in each line are separated by semicolons.

One such line look like this:
<user>; <rights>; <cfilesystem>; <cfile>; <mount point>; <key>

Explanation:

<user> :	Name of user	
<rights> :	this can be either '*' or 'U'.	
	'*'	owner of the container file (only the owner can grant 'U' rights to other users and he/she can delete the container file)
	'U'	normal user of the container file
<cfilesystem> :	filesystem inside the encrypted container file	
<cfile> :	path of the container file	
<mount point> :	defines where the container file will be mounted	
<key> :	cleartext key to decrypt the AES-encoded key contained on the USB stick	

Event-Notify Structure (in EBNF):

output	::= <crypt_event>[**unplug_event**][**plug_event**]</crypt_event>
unplug_event	::= <unplugged>{**device_u**}+</unplugged>
plug_event	::= <plugged>{**device_p**}+</plugged>
device_u	::= <device name=*DEVICE*></device>
device_p	::= <device name=*DEVICE*>{**file**}+</device>
file	::= <file>*FILE*</file>
DEVICE	::= device file (e.g. /dev/sda)
FILE	::= content of a file under ./crypt-loop

Programs in the detail

Secure communication – Detailed solution

As mentioned above OpenSSH is used for secure communication. The 'Crypto Manager' (p. 13ff) starts the 'USB Watch' program (p. 11ff) on the X-Terminal by the use of OpenSSH.

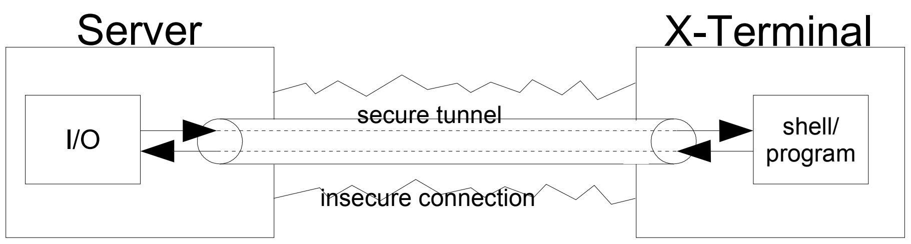

Figure 5 – Secure connection with OpenSSH

Figure 5 shows how OpenSSH is used. The server starts a shell or a program on the X-Terminal. All the I/O of the shell/program is being tunneled over the secure connection.
To achieve transparent login at the X-Terminal RSA key pairs without a pass phrase are used.

To generate such a key pair type the following:
```
ssh-keygen -t rsa -f <filename>
```

The public part of the key must be added to '/root/.ssh/authorized_keys' file on the X-Terminal.
The private part of the key must be saved in a secure place on the server.
By doing that everybody who gains access to the private key could login at the terminals as root. To prevent that, the line in the authorized_keys file needs to be modified. Adding command="<command>" at the beginning of the line only allows this command to be executed. Neither shell-login nor executing another command is than possible.

The 'Crypto Manager' is written in Perl. The wrapper-part forks (see code excerpt line 1/7). The child-part (line 8ff) executes OpenSSH/'USB Watch'. The parent-part (line 2ff) opens a pipe. The input of the pipe is the output of 'USB Watch'.

```
.
.
.
if(my $pid = open(CHILD, "-|")) {
  #parent

    # do stuff

  }
} else {
  #child
  die "ERROR: fork failed: $!" unless defined $pid;
  STDOUT->autoflush(1);

  exec("ssh","-i",$pub_keyfile,"-l","root",$ip_xterminal,$usb_watch_command)
    or die "ERROR: Couldn't exec 'ssh' : $!" ;

  exit;
}
.
.
.
```

USB Watch – Detailed solution

This part of the task is solved by two shell-scripts and a little C-program.

'USB Watch' supports every type of hardware which is recognized by Linux as USB-Storage device. It is not possible to distinguish between a USB-Stick, a USB-Flashcard reader or a mobile USB-Hard disk. This is because Linux displays all these devices as usb-mass-storage devices. For simplicity the program works with all of them with the limitation that only the first partition on such a device is scanned. This is ok in most cases, because standard USB-Sticks (which should work) only possess one partition.

'usb_loop.sh'
The 'Crypto Manager' executes, by the use of OpenSSH, the shell-script 'usb_loop.sh' on the X-Terminal. This script executes in a loop the shell-script 'usb_detect.sh' (see in detail further down).

```
1  #!/bin/sh
2
3  /opt/usb_crypt/usb_detect.sh
4
5  while ps ax | grep ssh | awk '{ print $1 }' | grep $PPID > /dev/null ; do
6    if /opt/usb_crypt/usb_wait /proc/bus/usb/devices; then
7      /opt/usb_crypt/usb_detect.sh
8    fi
9  done
```

The code above shows the script 'usb_loop.sh'. In line 3 the script 'usb_detect.sh' is initially executed to scan for already attached USB-devices. To avoid a 'busy wait' the C-program 'usb_wait' (see in detail further down) is called in every loop. Because the script 'usb_loop.sh' doesn't get terminated by OpenSSH the loop has the termination condition in line 5. In this line all PIDs of running OpenSSH connections are determined. These PIDs are compared to the parent PID of 'usb_loop.sh' (because it's parent is a OpenSSH process). As long as this parent PID exists the server connection still is working and therefore the loop continues.

'usb_wait'
The little C-program 'usb_wait' opens the file '/proc/bus/usb/devices'. This file is part of the usbdevfs. It contains information of all attached USB devices. It can only be accessed if it is mounted as mentioned above. A file handle is opened and monitored by the help of the syscall 'select'. The plugging and unplugging of a USB device is changing this file. 'usb_wait' recognizes this and terminates.

'usb_detect.sh'
The shell script 'usb_detect.sh' searches for plugged/unplugged USB-Storage devices. If such a device contains crypto information it is printed to stdout as a XML stream.

Linux handles USB-Storage devices as SCSI-devices. Information about USB-Storage devices is located under '/proc/scsi/usb-storage*'.
Such an entry e.g. looks like this with a 2.4 Kernel:

```
   Host scsi0: usb-storage
       Vendor: USB
      Product: Solid state disk
Serial Number: 212815D73E5FE714
     Protocol: Transparent SCSI
    Transport: Bulk
         GUID: 0ea06803212815d73e5fe714
     Attached: No
```

Relevant are only the first line and the attached entry in the last. The Kernel assigns a SCSI host device to every USB-Storage device by it's GUID (this GUID is generated by : <vendorid><productid><serialnr>). In the example hostnumber 0 (Host scsi0).
First all USB-Storage entries, which have set 'Attached: Yes', are parsed. These appropriate host numbers are stored.
These host numbers are assigned, by the kernel, to a SCSI generic device. The hostnumber is always the number of the generic device. E.g. host0 is /dev/sg0.
To be able to mount the plugged USB-Sticks the high level device assigned by the kernel needs to be found out. With the Tools sg_scan/sg_map from the package sg3utils [4] this assignment can be found out. E.g. host0 is /dev/sda.
Now that the assignment is known, it has to be checked if the device is mounted. If not, it is mounted and it's contents are searched for the directory '.crypt_loop'. If found and therein files exist, the device is listed in the '<plugged>' section in the event notify structure (described above).
If a USB-Storage device is unplugged the program compares all devices with status 'Attached: Yes' and the output of the 'mount' command. All devices which are unplugged right now are still listed in the 'mount' output. These are now unmounted and are added to the <unplugged> section in the event notify structure (described above).

The final output of this script could look like that:

```
<crypt_event>
  <unplugged>
    <device name=/dev/sda></device>
    <device name=/dev/sdc></device>
  </unplugged>
  <plugged>
    <device name=/dev/sdb>
      <file>[FILE CONTENTS]</file>
    </device>
  </plugged>
</crypt_event>
```

Crypto Manager – Detailed solution

Like it's name expresses this program is the core application. After being started at/by the login process it manages/supervises the other parts.
As mentioned above in the secure connection part, the Crypto Manager establishes an OpenSSH tunnel and starts the USB Watch program at the X-Client.
For every logged-in user a new instance of the Crypto Manager is started. Because multiple users should be able to work with the same crypto file and there is only a limited number of loopback devices, the same crypto files are assigned to one loopback device. The Crypto Managers therefore must communicate with each other to know which loopback device is assigned to which crypto file. This is achieved by a global assignment file. To prevent inconsistencies in that file, the multiple programs writing to it have to gain exclusive access. To ensure this, a Crypto Manager has to do a try-lock on that file if it wants to read/write from/to that file. If the assignment file is locked already by another Crypto Manager the current process is suspended until it itself can gain exclusive access.

```
 .
 .
 .
 1  CHILD->blocking(0);
 2
 3  my $run = 1;
 4
 5  my $rin = '';
 6  my $rout='';
 7
 8  my $buffer = "";
 9  vec($rin, fileno(CHILD), 1) = 1;
10
11  while($run) {
12    my $h_parse_ret;
13
14    my $count = select($rout=$rin, undef, undef, undef);
15
16    if($count) {
17      if(vec($rout, fileno(CHILD),1)) {
18      }
19    }
20    if($buffer eq '') {
21      print "SSH-Connecting lost ... \n";
22
23      sudden_death(); # cleanup, umount/losetup -d/rm -r
24
25      $run = 0;
26    }
27    if($buffer =~ m/^\s*<crypt_event>\s*(.*)\s*<\/crypt_event>\s*$/s) {
28      my $rest = $1;
29
30      $h_parse_ret = parse_event($rest);
31    } else {
32      die "ERROR: Never should be here!!\n";
33    }
34
35    work_event($h_parse_ret);
36
37    $buffer = "";
38  }
 .
 .
 .
```

The listing above shows in slightly shortened form how the main-loop works. The program waits (l. 14) for pending input from USB Watch. Normally 'select' waits for 'eof' (end of file) until it returns. Because USB Watch continuously sends data over a pipe no 'eof' can occur. To prevent 'select' to wait infinitely the input file handle is set to nonblocking mode (l. 1)
If 'select' returns, the available data is read into a buffer. Should the buffer be of zero-length this means, that there is a problem with the client (USB Watch) connection. The cleanup routine is called immediately and the Crypto Manager terminates.

Otherwise (buffer length greater zero) it is checked/parsed (l. 30 - described in detail later) if the buffer contains valid crypto event data. If it doesn't the cleanup routine is called and the program terminates.
The parsed data now is handled by 'work_event' (l. 35). This routine handles the data structure previously parsed. This means mounting/unmounting the appropriate crypto file systems (described in detail later).

'parse_event' gets all the data from 'usb_watch'. The structure sent is explained above (Used data/file structures – Event notify structure)
This structure is parsed and transformed into a perl-hash (see Perl Documentation for more). The following shows an example of such a hash.

```
hash {
      'plugged'   => {
                        '/dev/sda1' => {
                                        [FILE1],
                                        [FILE2]
                                       }
                       }
     'unplugged' => {
                        '/dev/sdb1'
     }
```

On top it is a hash containing two keys : plugged and unplugged. These two keys are again a reference to another hash containing as keys devices. These device keys are references (in case of the plugged event) to an array of scalar values containing the base64 encoded files.

'work_event' starts with the found *plugged* events.
For each new device one or more password files exist. These were sent base64 encoded (see above – Used data/file structures).
After decoding the sent file, the master config file is searched. First the given username from the password file is compared. If there is a match, 'work_event' tries to decode the password from the stick with the password found in the master config file. If decoding doesn't succeed the next entry of the master config file is processed.
On success the function 'try_mount' is called. The following pseudo code shows it's functionality.

```
.
try_mount($input) {
  try_lock($mutex_file)
  foreach $entry in $mutex_file {
    if $input->enc_file eq $entry->enc_file {
      if $input->mnt_pt eq $entry->mnt_pt {
        warning('WARNING: Your mount point is already in use !')
        close($mutex_file)
        return
      }
      mount($entry->loopdev, $input->enc_file, $input->mnt_pt)
      update($mutex_file)
      update($global)
      close($mutex_file)
      return
    }
  }
  $loopdev=find_free_loopdev();
  assign($loopdev, $input->pw);
  mount($loopdev, $input->enc_file, $input->mnt_pt)
  update($mutex_file)
  update($global)
  close($mutex_file)
  return
}
.
```

The previous found data from the master config file is passed to 'try_mount'. Now the already explained global assignment file (from now on called mutex file) is locked. Then it is checked if the container file is already assigned to a loopback device. If it is, that loopback device is remounted, the mutex file and a global data structure which contains the same data as the mutex file (only for this process) are updated. At last the mutex file is closed/unlocked.
If the container file isn't assigned yet, a new loopback device is assigned, the container mounted, the mutex file and the global data structure are updated. Finally the mutex file is closed/unlocked.

Now 'work_event' continues with the found *unplugged* events.
The pseudo code below shows how the function 'work_event_unplugged' works.

```
.
.
.
work_event_unplugged($input) {
  try_lock($mutex_file)
  foreach $entry in $mutex_file {
    foreach $dev in $input {
      if defined $global->{$dev} {
        foreach $loop in $global->{$dev} {
          foreach $mnt_pt in $loop {
            if $entry->loopdev eq $loop {
              umount($mnt_pt)
              try_free_loopdevs()
              update($mutex_file)
              update($global)
            }
          }
        }
      }
    }
  }
  close($mutex_file)
  return
}
.
.
.
```

The *unplugged* part of the data structure generated by 'parse_event' is passed as argument.
The mutex file is locked. Then a few things have to be checked. First, for all the devices given in the input, the assigned loopback devices have to be looked up in the global data structure. Second, every loopback device can have one or more mount points. These have to be unmounted. If a loopback device is no more in use it is freed. Third the mutex file and the global data structure are updated. At last the mutex file is closed/released.

Tool

To get all running comfortable there is a script that sets up a new container file, all needed entries in the master configuration file and the file to be used on the usb-stick.
The script is called 'create_new_container.pl'.

```
USAGE: create_new_container.pl -e <enc> -k <keybits> -fs <cfs> -co <c_opts> -mo <m_opts> -s
       <cfs_size> -u <user> -mnt <mnt_pt>
```

With the <enc> and <keybits> options the encoding and the used key length of the container file are set. <cfs> and <c_opts> define what filesystem should be generated in the container file and what filesystem options should be used in generation. <m_opts> are the options that should be used when mounting the container file. <cfs_size> defines the size of the container file in MB. <user> defines the owner of the container file. Finally <mnt_pt> defines where the container file will be mounted.

VIII.Differences between 2.4 and 2.6 Kernel

Since when the work on this project started, the new 2.6 Kernel has been released. A lot of problems which had to be solved using a 2.4 Kernel don't exist anymore.
Now no patch has to be applied to the Kernel. The cryptoloop device driver and all the cryptographic functions are now part of the official release.
Additionally a new pseudo-filesystem, the sysfs, has been added. With the sysfs the scsi-generic tools become obsolete for this project. The assignment problem can now easily be parsed from the data provided by the sysfs.

IX.Conclusion and future work

In principle the program runs. However various restrictions and deficiencies have to be considered.
A problem that one should consider, is that with switching to a new kernel version the encoding algorithms of the CryptoAPI and/or the cryptoloop module may make the generated and used container files inoperative. During the 'Systementwicklungsprojekt' once the encryption algorithms of the CryptoAPI and once the cryptoloop module were changed.
A further problem is the stability of the Linux USB drivers. By frequent plugging and unplugging of USB devices (while they are accessed) one can very easily irreversible crash the Linux USB drivers. The only possibility to get the USB drivers back working is to reboot the system.
There are also some deficiencies in using USB sticks for authentication. USB sticks do not have unique recognition characteristics. Each USB device supplies a vendorid, a productid and a serial number. These values however are no reliable recognition characteristics. Although vendorid and productid are indicated correctly for each USB device, many manufacturers unfortunately saved money and didn't add a correct serial number. Often this consists completely of zeros or ones.
To be able to surely assign the authentication data to one USB stick it should have a unique serial number. So one could combine the data on the USB stick with it's serial number and than digitally sign this tuple.
By that USB stick and it's containing data form a clear unit. The data could thus not be duplicated. On another USB stick the data would be useless.
Thus one would have the greatest possible security. Only the theft of the USB stick or the public signing key still represent danger.
A further deficiency is given with the cryptoloop implementation. When setting up an encoded partition (all the same whether it concerns a whole block device or just a container file on a hard disk) a pass phrase is assigned.
With mounting of a encrypted partition gets assigned to a loopback device after the pass phrase is entered. Each user with administrator rights or the right to access this device can now open it, without the need to know the pass phrase, and read all the containing data in clear.
Finally it remains saying that USB sticks are not or only badly suitable for authentication due to the missing unique characteristics.
Encoded file systems are quite applicable under Linux. However one should be cautious with the change of the kernel version. Because encoded file systems under Linux, once their pass phrase is entered, can be accessed by everyone with the appropriate rights, they are mainly suitable to save the data from physical access or theft.

X.Bibliography

[1] Computer Security Resource Center – AES Seite
http://csrc.nist.gov/CryptoToolkit/aes/

[2] Christian Kruse: Die sichere Passwort-Wahl
http://selfaktuell.teamone.de/artikel/gedanken/passwort/index.htm

[3] secpass - Secure random password generator
http://kiwispam.sourceforge.net/kiwi-2.0.00/tools/secpass

[4] The Linux SCSI Generic (sg) Driver
http://www.torque.net/sg/

[5] The Linux USB sub-system
http://www.linux-usb.org/USB-guide/book1.html

[6] Universal Serial Bus: Architektur, Installation, Konfiguration, Praxis, Programmierung
Linux-Magazin 10/2000

Appendix: Installation

The tar-file 'usb_crypt.tar.gz' needs to be unpacked in the directory '/opt' with the command

```
tar xzf usb_crypt.tar.gz
```

on the server and the X-terminals.
Now produce the directory '/mnt/usb-storage' on the X-terminals.
Change to the directory '/opt/usb_crypt/key' and create with the command

```
ssh-keygen -t rsa client_key
```

a ssh key (leave the passphrase empty!). Use

```
cat client_key.pub >> /root/.ssh/authorized_keys
```

to append the public part of the key for usage on the X-terminals.
So that the Crypto Manager can log-in later on the Clients, one must log-in manually now once with

```
ssh -f client_key root@<client>
```

This has to be done so that the scripts later on can logon interactivly. When a user logs in the first time a few questions need to be answered. The second and all further logins then work without any further questions.
Does that have functioned error-free on all X-terminals, one, for safety reasons should add the following at the beginning of the keyline

```
'command="/opt/usb_crypt/usb_watch"'
```

By doing that, it is only possible to execute this one command. Now one should produce a container file as described above (see p. 15). On the server thereby the file 'new_stick_data' is created. This file should be renamed and copied on the USB-Stick in the directory '<path>/.crypto_info'.
By running

```
/opt/usb_crypt/crypto_manager <ip-client>
```

now on the server the dynamic/transparent usage of the just prepared USB-Stick on that X-terminal is now possible.