Title

Cryptographic key recovery from Linux memory dumps

Doc. ID	Version
	A
Author	Saved
Torbjörn Pettersson	2007-04-19
Distribution	
Archive, CCC	

Contents

1 Preface

1.1 Purpose and readers

The purpose of this document is to investigate in detail how to perform cryptographic key recovery from memory dumps on the GNU/Linux platform.

Target group for this document is anyone interested in cryptography and computer forensics on the GNU/Linux platform.

1.2 Revision notes

A	2007-04-24	T.P.	Final version

1.3 References

[1]	Linux kernel source	http://www.kernel.org/
[2]	TrueCrypt	http://www.truecrypt.org/
[3]	TPM introduction	http://wikipedia.org/wiki/Trusted_Platform_Module
[4]	Unlocking File Vault	http://crypto.nsa.org/vilefault/23C3-VileFault.pdf
[5]	Suspend2	http://www.suspend2.net/
[6]	Hacking by Firewire	http://md.hudora.de/presentations/#firewire-cansecwest
[7]	Util-linux	ftp://ftp.kernel.org/pub/linux/utils/util-linux/
[8]	cryptoloop	http://en.wikipedia.org/wiki/Cryptoloop
[9]	dm-crypt	http://www.saout.de/misc/dm-crypt/

1.4 Scope

The document only describes key recovery from Linux systems. Linux provides a number of different disk encryption solutions, key recovery procedures for the two native solutions will be detailed; cryptoloop and dm-crypt.

1.5 Background

In the last few years cryptography related laws in the US have undergone significant changes allowing for inclusion of an advanced cryptographic abstraction layer in the stock Linux kernel [1]. Other open source solutions such as TrueCrypt [2] also provide strong disk encryption functionality with minimal effort on multiple platforms.

The skills and amount of work required to set up disk encryption is often negligible when compared to the comfort and security provided by these systems. As a result, it's very common for hackers, pirates, dissidents and other people with legit paranoia to encrypt all or part of their hard drives.

The ease and popularity of these tools provide forensic analysts with a formidable challenge. Computer forensics is an ever-evolving science and it requires significant efforts to keep pace in the technological arms race. When disk encryption is done right it is virtually impossible to break, but as with any security function it is never stronger than its weakest link.

2 Memory imaging techniques

Even though this paper isn't about forensic memory acquisition techniques it might be useful to the reader with a short introduction to different approaches to collecting the memory image being analyzed in the examples throughout this paper.

2.1 Physical memory pseudo devices

Modern operating systems often provide a way for users to access physical memory. Most Unix platforms provide `/dev/mem` and Microsoftish flavors provide the pseudo-device `\\.\PhysicalMemory` that can be accessed using tools such as dd.

2.2 Hibernation files

Since laptops have a limited battery life they often provide the option to store the content of memory to disk in order to provide prolonged suspend beyond the capabilities of the built in battery. During this process a complete image of the memory will be written to the laptop disk including any cryptographic keys. Most modern operating systems supports some type of hibernation, on Mac OS X its called *safe sleep* and on the Linux platform it is provided by a set of patches called *Suspend2* [5].

Due to architecture of modern computers it is difficult to encrypt the hibernation data without specialized hardware such as a TPM-chip [3]. Mac OS X [4] does support encrypting the safe sleep file but stores the cryptographic keys on the disk and thus making the encryption next to pointless.

2.3 Firmware breakpoints

Some type of platforms, notably the Sun Microsystems Sparc platform, allows for the user at the console to press a magic key-combo to suspend the operating system and hand over control to the built in firmware.

2.4 Virtualization

When the system is running in a virtualization environment it is often easy to get access to a copy of the virtual machines memory by simply pausing the operating system and suspending it to disk.

2.5 Firewire

The Firewire standard, IEEE-1394, provides for Direct Memory Access (DMA) to the physical memory of the connected host. This can be used to image the entire RAM over the firewire-interface without interfering with the operating system [6].

3 Cryptoloop

3.1 Background

Cryptoloop is the oldest disk encryption architecture in the Linux kernel environment. Cryptoloop started out as a set of patches for the Linux kernel distributed by kerneli.org and eventually evolved (different author) into the CryptoAPI in the 2.5.x-series of Linux kernels as the laws and regulations surrounding cryptography changed. As the name suggest it hooks into the loop-back mounting feature of Linux kernel that allows for a file to be represented and handled as a block device. Files or existing block devices can be stacked with a cryptographic function and presented to the user as a fully functional block device containing a normal files system of choice.

Cryptoloop has a number of problems and vulnerabilities and has therefore officially been deprecated from the 2.6-line of kernels. Cryptoloop is replaced by dm-crypt as the 'official' disk encryption solution for Linux.

3.2 Key storage structure

The keys for cryptoloop is stored in the loop-struct defined in `include/linux/loop.h`.

```
struct loop_device {
        int             lo_number;
        int             lo_refcnt;
        loff_t          lo_offset;
        loff_t          lo_sizelimit;
        int             lo_flags;
        int             (*transfer)(struct loop_device *, int cmd,
                                    struct page *raw_page, unsigned raw_off,
                                    struct page *loop_page, unsigned loop_off,
                                    int size, sector_t real_block);
        char            lo_file_name[LO_NAME_SIZE];
        char            lo_crypt_name[LO_NAME_SIZE];
        char            lo_encrypt_key[LO_KEY_SIZE];
        int             lo_encrypt_key_size;
        struct loop_func_table *lo_encryption;
        __u32           lo_init[2];
        uid_t           lo_key_owner;   /* Who set the key */
        int             (*ioctl)(struct loop_device *, int cmd,
                                 unsigned long arg);

       struct file *   lo_backing_file;
        struct block_device *lo_device;
        unsigned        lo_blocksize;
        void            *key_data;

        gfp_t           old_gfp_mask;

        spinlock_t              lo_lock;
        struct bio              *lo_bio;
        struct bio              *lo_biotail;
        int                     lo_state;
        struct completion       lo_done;
        struct completion       lo_bh_done;
        struct semaphore        lo_ctl_mutex;
        int                     lo_pending;

        request_queue_t         *lo_queue;
};
```

3.3 The keying process

Keying for Cryptoloop is done through the `LOOP_SET_STATUS` ioctl. For most parts, either `losetup` or `mount` will call this ioctl when the user specifies an encryption algorithm and gives a password.

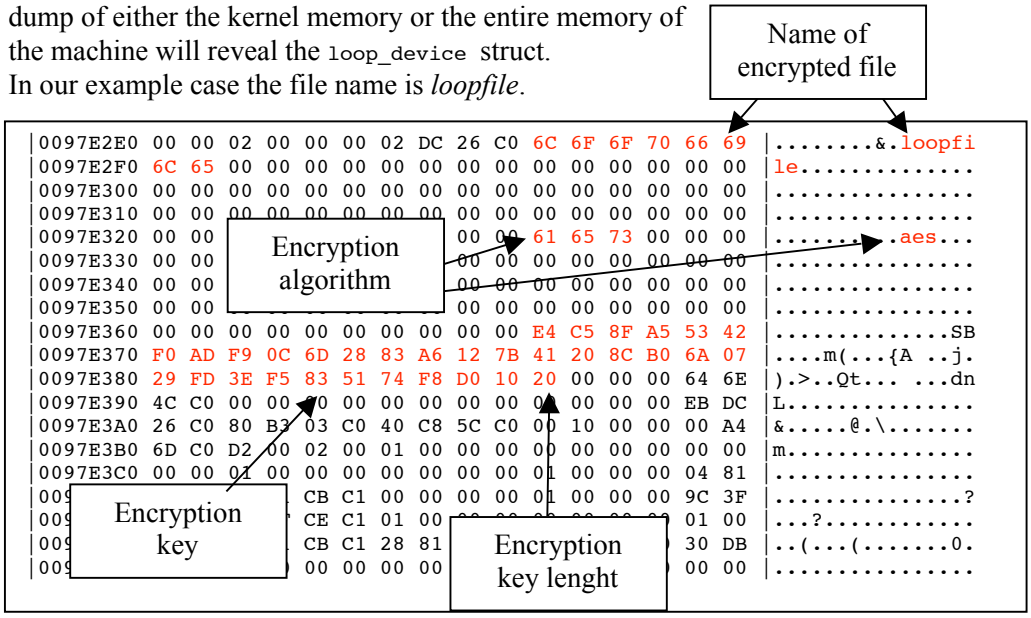
`losetup` and `mount` are distributed as part of the util-linux package [7], but there exists a number of different patches that affects how keying is performed. The pristine util-linux package doesn't provide any hashing or salting of the password before presenting it as the encryption key to the kernel. Debian and perhaps other distributions choose to patch util-linux in order to add hashing of the given password before using it as the encryption key for the device.

From a key recovery standpoint the way that the key got into kernel memory makes little difference.

3.4 Finding the key

It is surprisingly simple to locate the key used by cryptoloop in kernel memory. As can be seen in the `loop_device` struct above, there is only a single key stored between the name of the encrypted file and the name of the crypto. All of the fields are of a fixed size and will be null-padded when not filled.

A simple search with a hex editor for the filename of the encrypted file in the memory dump of either the kernel memory or the entire memory of the machine will reveal the `loop_device` struct.
In our example case the file name is *loopfile*.




```
|0097E2E0 00 00 02 00 00 00 02 DC 26 C0 6C 6F 6F 70 66 69 |........&.loopfi
|0097E2F0 6C 65 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |le..............
|0097E300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|0097E310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|0097E320 00 00                                     00 00 61 65 73 00 00 00 |........aes...
|0097E330 00 00                                     00 00 00 00 00 00 00 00 |................
|0097E340 00 00                                     00 00 00 00 00 00 00 00 |................
|0097E350 00 00                                     00 00 00 00 00 00 00 00 |................
|0097E360 00 00 00 00 00 00 00 00 00 00 E4 C5 8F A5 53 42 |..............SB
|0097E370 F0 AD F9 0C 6D 28 83 A6 12 7B 41 20 8C B0 6A 07 |....m(...{A ..j.
|0097E380 29 FD 3E F5 83 51 74 F8 D0 10 20 00 00 00 64 6E |).>..Qt... ...dn
|0097E390 4C C0 00 00 00 00 00 00 00 00 01 00 00 00 EB DC |L...............
|0097E3A0 26 C0 80 B3 03 C0 40 C8 5C C0 00 10 00 00 00 A4 |&.....@.\.......
|0097E3B0 6D C0 D2 00 02 00 01 00 00 00 00 00 00 00 00 00 |m...............
|0097E3C0 00 00 01 00 00 00 00 00 00 00 01 00 00 00 04 81 |................
|009                CB C1 00 00 00 00 01 00 00 00 9C 3F |...............?
|009                CE C1 01 00                     01 00 |...?............
|009                CB C1 28 81                     30 DB |..(...(.......0.
|009                00 00 00 00                     00 00 |................
```

Example from a Debian machine using a 256 bit hash-value as encryption key for AES.

The example below uses twofish instead of aes and the key isn't hashed before it's loaded into kernel memory, therefore the password can be read in clear text.

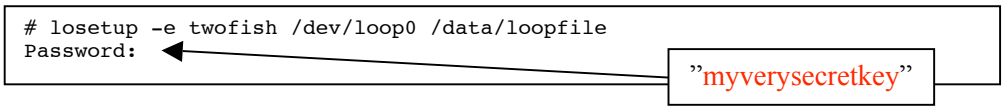
```
|0137C2F0 26 C0 2F 72 6F 6F 74 2F 73 65 63 72 65 74 73 74 |&./root/secretst
|0137C300 75 66 66 00 00 00 00 00 00 00 00 00 00 00 00 00 |uff.............
|0137C310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|0137C320 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|0137C330 00 00 74 77 6F 66 69 73 68 00 00 00 00 00 00 00 |..twofish.......
|0137C340 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|0137C350 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|0137C360 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|0137C370 00 00 6D 79 76 65 72 79 73 65 63 72 65 74 6B 65 |..myverysecretke
|0137C380 79 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |y...............
|0137C390 00 00 20 00 00 00 64 6E 4C C0 00 00 00 00 00 00 |.. ...dnL.......
```

Example from a machine using a clear text key as encryption key for Twofish.

3.5 Using the collected key

3.5.1 Clear text key for cryptoloop

In the case where the key can be read in clear text the reuse of the key is simple.



3.5.2 Hashed key for cryptoloop

Due to a bug in util-linux it is impossible to input keys longer than 31 bytes, this problem is explained in detail in **Appendix B**. Since hashed 256bit keys always are 32 bytes it’s not possible to input these keys without applying the patch from Appendix B. Another problem is that keys are always handled as strings by util-linux and it is therefore impossible to input keys that contain a zero (0) byte. If the hashed key contains a null-byte specialized tools needs to be developed to hand over the key correctly to the kernel.

One of the first requirements is to write the key in binary format to a file. This can be done using a hex editor, but one of the simplest methods is to use perl as in the example below.

```
# perl -e 'print "\xE4\xC5\x8F\xA5\x53\x42\xF0\xAD\xF9\x0C\x6D";
           print "\x28\x83\xA6\x12\x7B\x41\x20\x8C\xB0\x6A\x07";
           print "\x29\xFD\x3E\xF5\x83\x51\x74\xF8\xD0\x10"'
      > /tmp/key.bin
```

Perl can be used to easily convert a hexadecimal key to binary form

`losetup` has an option to read the key from a file descriptor (`-p`). Once `losetup` has been patched to accept 32 byte key strings this option can be used to input the binary key from standard input (file descriptor zero).

```
# cat /tmp/key.bin | losetup –e aes –p0 /dev/loop0 /data/loopfile
```

losetup has to be patched in order to accept 32 byte keys

4 dm-crypt

4.1 Background

dm-crypt [9] is the replacement for cryptoloop. It uses the built in device-mapper subsystem in the Linux 2.6-series of kernels to stack a transparent cryptographic function between the user and the block device. It uses the CryptoAPI in much the same way as cryptoloop but is considered to be cryptographically superior. One of the main differences apart from dm-crypt being a better solution is that dm-crypt only supports block-devices. When files are used as containers they first need to be converted to block-devices using the loop-subsystem.

4.2 Key storage structure

The keys used by dm-crypt are stored in the `crypt_config` struct defined in `dm-crypt.c`

```
struct crypt_config {
        struct dm_dev *dev;
        sector_t start;

        /*
         * pool for per bio private data and
         * for encryption buffer pages
         */
        mempool_t *io_pool;
        mempool_t *page_pool;

        /*
         * crypto related data
         */
        struct crypt_iv_operations *iv_gen_ops;
        char *iv_mode;
        void *iv_gen_private;
        sector_t iv_offset;
        unsigned int iv_size;

        struct crypto_tfm *tfm;
        unsigned int key_size;
        u8 key[0];
};
```

4.3 The keying process

Keying for dm-crypt is done using the tool cryptsetup. A modified version of cryptsetup called cryptsetup-LUKS adds the feature of storing all the encryption options in a special luks-header at the start of the encrypted device. This makes it easier to identify encrypted devices as well as the encryption algorithm and the key length. It is difficult but perfectly possible to setup dm-crypt using only the native `dmsetup` tool. For the purpose of extracting and reusing the encryption key from kernel memory the difference is minute.

4.4 Finding the key

The `crypt_config` struct doesn't contain any really good information that can be used in a search for the information. This makes it a bit tricky to actually find the key in a memory dump. The `crypt_config` struct contains only pointers to most of the interesting information. In some cases it might be possible to dereference the pointers from the device struct and find the key manually using a hex-editor but in most cases this will prove to be very difficult.

In order to do a binary search for the `crypt_config` struct some assumptions have to be made about the values of each of the struct members.

The majority of the struct members are pointers to other parts of the kernel memory; `dev`, `io_pool`, `page_pool`, `iv_gen_ops`, `iv_mode`, `iv_gen_private` and `tfm`. Since all memory used by the Linux kernel is located above the address 0xC0000000 it can be assumed that all of these pointers have a value of 0xC0000000 or above.

The `start` member can have any value between 0 and the size of the target destination.

The `iv_offset` will be zero (0) during normal operation.

The value of `iv_size` will be either 16 or 32 depending on the encryption algorithm chosen by the user.

For the purpose of this paper the `key_size` is assumed to be 16 or 32 bytes even though it might in some cases be 8 (64 bit) or 24 (192 bit).

Using this information rules can be created when searching the memory dump for cryptographic keys. The entire key-extraction program can be found in **Appendix A.**

```
if(
            (void *) cr->io_pool        > (void *) 0xc0000000 &&
            (void *) cr->tfm            > (void *) 0xc0000000 &&
            (void *) cr->dev            > (void *) 0xc0000000 &&
            (void *) cr->io_pool        > (void *) 0xc0000000 &&
            (void *) cr->page_pool      > (void *) 0xc0000000 &&
            (void *) cr->iv_gen_ops     > (void *) 0xc0000000 &&
            (void *) cr->iv_mode        > (void *) 0xc0000000 &&
            (void *) cr->iv_gen_private > (void *) 0xc0000000 &&
            (cr->key_size == 16 || cr->key_size == 32) &&
            (cr->iv_size  == 16 || cr->iv_size  == 32) &&
            cr->iv_offset == 0
  )
printkey(cr->key,cr->key_size)
```

4.5 Using the collected key

4.5.1 cryptsetup

When the user is using dm-crypt and loads the keys using `cryptsetup` or manually using `dmsetup` the key recovery and reuse is very straightforward.

```
# ./keysearch memorydump
keylenght: 256
key: 4C6B8BE4311D556FDE82381C94855827D88ADE4344C240B7999C1D7254D67598
```

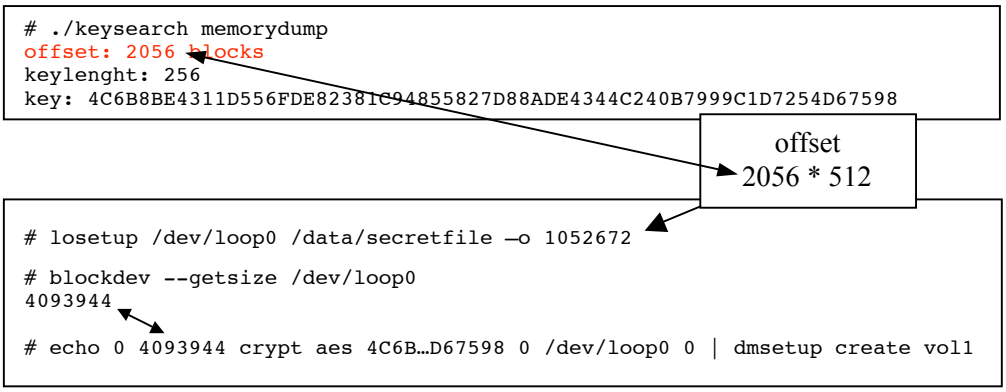
```
# losetup /dev/loop0 /data/secretfile

# blockdev --getsize /dev/loop0
4096000

# echo 0 4096000 crypt aes 4C6B…D67598 0 /dev/loop0 0 | dmsetup create vol1
```

The only real problem is the encryption algorithm; in this test case it's already known that the users has chosen to use AES. In other cases it might be apparent from the key-length or from the LUKS-header if available. A worst-case scenario is to try to brute force the chosen encryption algorithm.

4.5.2 cryptsetup-luks

The difference between `cryptsetup` and `cryptsetup-luks` is that the luks-version stores cryptographic information in a special LUKS-header at the beginning of the cryptographic container. `cryptsetup-luks` uses the offset feature of dm-crypt, which makes it a bit trickier to access the volume manually. On the other hand, the luks-header provides valuable information about the chosen cryptographic algorithm as well as for easy identification of cryptographic containers.

When running the `keysearch` program the offset is printed if it is set.



The option to `losetup` simply offsets the beginning of the loop-device past the LUKS-header just as cryptsetup-luks would do. Since the offset is printed in 512-byte blocks and the offset option for `losetup` is in bytes the provided value has to be multiplied by 512. The size of the device also changes slightly due to the LUKS-header.

5 Conclusions

5.1 Key recovery

The obvious conclusion that can be drawn is that key recovery is surprisingly easy. Once an attacker, in our case a forensic investigator, has gained access to an image of the memory the keys can be easily found, extracted and reused to gain access to the encrypted material.

Information about the chosen cryptographic subsystem, tools and algorithm needs to be gathered using traditional forensic methodologies prior to key recovery.

5.2 Defensive mechanisms

The only viable defence against memory based key recovery is toughening the system against known memory acquisition techniques. The firewire system available in most modern laptops needs to be disabled as well as any hibernation support. It is obvious that from a security standpoint it is a significant risk to run a system that handles sensitive key material in any type of virtualization environment.

Strong passwords and screensavers with password protection will in many cases make it impossible or difficult to gain access to the memory content even if access to the hardware is gained physically.

6 Appendix A

This program searches a memory dump from a Linux machine and extracts any cryptographic keys used to encrypt devices using dm-crypt.

```
/* © Torbjörn Pettersson 2007*/

#define __KERNEL__ /* Only needed to enable some kernel-related defines */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <linux/mempool.h>

struct crypt_config
{
   struct dm_dev *dev;
   sector_t start;

   mempool_t *io_pool;
   mempool_t *page_pool;

   /* crypto related data */
   struct crypt_iv_operations *iv_gen_ops;
   char *iv_mode;
   void *iv_gen_private;
   sector_t iv_offset;
   unsigned int iv_size;


  struct crypto_tfm *tfm;
   unsigned int key_size;
   u8 key[0];
} __attribute__ ((packed));

int keysearch(char *mem, int size)
{
   int i,j;
   struct crypt_config *cr;

   for(i = 0; i < (size - sizeof(struct crypt_config)); i++,mem++)
     {
        cr = (struct crypt_config *) mem;

        if(
           (void *) cr->io_pool        > (void *) 0xc0000000 &&
           (void *) cr->tfm            > (void *) 0xc0000000 &&
           (void *) cr->dev            > (void *) 0xc0000000 &&
           (void *) cr->io_pool        > (void *) 0xc0000000 &&
           (void *) cr->page_pool      > (void *) 0xc0000000 &&
           (void *) cr->iv_gen_ops     > (void *) 0xc0000000 &&
           (void *) cr->iv_mode        > (void *) 0xc0000000 &&
           (void *) cr->iv_gen_private > (void *) 0xc0000000 &&
           (cr->key_size == 16 || cr->key_size == 32) &&
           (cr->iv_size  == 16 || cr->iv_size  == 32) &&
           cr->iv_offset == 0
           )
          {
             if(cr->start > 0)
               printf("offset: %ld blocks\n",
                       (unsigned long int ) cr->start);
             printf("keylenght: %d\n",
                     (cr->key_size * 8));
             printf("key: ");
             for(j = 0; j < cr->key_size; j++)
               printf("%02X",cr->key[j]);
             printf("\n");
          }
     }
   return(0);
}
```

```
int main(int argc, char **argv)
{
   int fd;
   char *mem = NULL;
   struct stat st;

   if(argc < 2)
     {
        printf("Usage: %s [memory dump file]\n",argv[0]);
        exit(-1);
     }

   if(stat(argv[1],&st) == -1)
     {
        perror("stat()");
        printf("Failed to stat %s\n",argv[1]);
        exit(-1);
     }

   fd = open(argv[1],O_RDONLY);
   if(fd == -1)
     {
        perror("open()");
        printf("Failed to open %s\n",argv[1]);
        exit(-1);
     }

   mem = mmap(0,(int)st.st_size, PROT_READ, MAP_SHARED, fd, 0);
   if(mem == ((void *) -1))
     {
        perror("mmap()");
        exit(-1);
     }

   (void)keysearch(mem,(int)st.st_size);

   return(0);
}
```

7 Appendix B

7.1 Util-linux

Util-linux has several known problems that for unknown reasons haven't been addressed in the official version; two of them make it difficult to reuse recovered keys.

7.1.1 String keys

The pristine version of util-linux handles keys as strings and not as binary data. This has several drawbacks from a cryptographic perspective, specifically it reduces the key space since it is impossible to input specific characters, notably the NULL-character is used represent string end and thereby terminate further key processing. It is quite likely that several other characters such as line-feed and carriage return wont be correctly processed either and thereby further reduce the key space.

7.1.2 31 byte keys

Due to a fencepost error all keys are NULL-terminated at the thirty-first character, this makes it impossible to input 32byte (256bit) keys and in practice further reduces the possible key space by 8 bits.

As can be seen in `xstrncpy.c` the function `xstrncpy()` will overwrite the `dest`-variable with NULL at n-1.

```
/* NUL-terminated version of strncpy() */
#include <string.h>
#include "xstrncpy.h"

/* caller guarantees n > 0 */
void
xstrncpy(char *dest, const char *src, size_t n) {
        strncpy(dest, src, n-1);
        dest[n-1] = 0;
}
```

The call made from `lomount.c` gives `LO_KEY_SIZE` (defined to 32 in `loop.h`) as argument to `xstrncpy()` and the result is a NULL-byte being written as the last byte in the key independent of the key-length the user specified.

```
xstrncpy(loopinfo64.lo_encrypt_key, pass, LO_KEY_SIZE);
```

7.1.3 Patches

There are several patches to resolve these issues, and for most users it's never an issue since their GNU/Linux distribution vendor often patches util-linux for them. But when binary keys are needed, such as when inputting a recovered pre-hashed key util-linux needs to be patched. One such patch written by Fruhwirth Clemens is available at http://clemens.endorphin.org/patches/util-linux-key-trunc-fix.diff and it enables the use of binary 32 byte keys with `losetup` and `mount`.