Mouting encrypted WD drives in linux

Thomas Kaeding (thomas.a.kaeding@gmail.com)
last modified 20170205

The purpose of this document is to explain how to mount an encrypted WD drive in a linux system, after the drive has been removed from its enclosure and the USB-to-SATA board has been removed. It must also be placed in a new enclosure that does not use hardware encryption or installed into a desktop and connected to an SATA port. These instructions assume that you can use a linux system and know how to find the terminal. All commands in this document are meant to be used in the terminal. We also assume that you know how to use the sudo command, and that you assume the risks of damaging your system or data.

If you use a new external enclosure, be careful that it presents a single drive to your system. Some break drives over 2TB into multiple drives of 2TB each.

First, determine where your system puts its device file for the drive. Look for an entry in /proc/partitions that is a single disk without partitions. For example, you might see the line

```
   8       32 3907018584 sdc
```

without any lines for sdc1. Check that you have found the right one with the command

```
sudo file -s /dev/sdc
```

If you see

```
/dev/sdc: data
```

and not some information about MBR or filesystems, then you probably have it. If you have other encrypted devices on your system, be careful that you indeed have the correct one.

We are going to be creating a few files along the way. Make a directory for them and enter it:

```
mkdir wd
cd wd
```

Did you set a password for the drive when it was in the original enclosure? If so, you need to generate the key encryption key (KEK) from it. Copy the code from Appendix A into a file called wd_kdf.sh and make it executable:

```
chmod +x wd_kdf.sh
```

Generate the KEK. For example, if the password was “mypassword”:

```
./wd_kdf.sh mypassword > kek.hex
```

If you did not set a password for the drive, then use the standard KEK (pi) and copy it into a file:

```
echo 03141592653589793238462643383279fcebea6d9aca7686cdc7b9d9bcc7cd86 > kek.hex
```

How to extract the disk encryption key (DEK) and set up the decryption filter is specific to which encryption chip is on the USB-to-SATA board.

JMicron JMS528S chip

Read the keyblock from the end of the disk. The location of this block depends on the size of your disk:

50 0GB	976769056
750 GB	1465143328
1 TB	1953519648
2 TB*	3907024928
3 TB	5860528160
4 TB	7814031392

(Note: the location for the 2TB drives is unverified.)

So, for example, if you have a 4TB disk at sdc, use this command:

```
dd if=/dev/sdc bs=512 skip=7814031392 count=1 of=kb.bin
```

Check to see that you have indeed obtained the keyblock by doing a hexdump and look for “WDv1”:

```
hexdump -C k0.bin
```

```
00000000  57 44 76 31 b3 db 00 00  00 b8 bf d1 01 00 00 00  |WDv1³Û...¸¿Ñ....|
00000010  03 00 00 00 00 00 f0 00  00 00 00 00 00 00 00 00  |......ð.........|
00000020  01 00 00 00 00 00 46 50  00 00 00 00 00 00 00 00  |......FP........|
00000030  00 02 ff 00 00 00 00 00  00 00 00 00 00 00 00 00  |..ÿ.............|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  20 00 3a 6a 00 00 00 01  00 00 00 00 57 44 76 31  | .:j........WDv1|
00000060  09 f8 45 57 df 43 28 50  2c 9e 4c 92 a0 93 b1 ed  |.øEWßC(P,.L. .±í|
00000070  1c 7e a7 1a 2a a5 8f 58  f5 06 c1 b5 6b 26 e7 18  |.~§.*¥.Xõ.Áµk&ç.|
00000080  5f d8 6e 2d 42 92 fe 5b  06 bc 30 b4 65 0f 87 b6  |_øn-B.þ[.¼0´e..¶|
```

Now for the hardest part: we need to build a new encryption module for the kernel. This module merely reverses the order of each 16-byte block. You will need to install all the necessary packages for kernel development. The C code for the new module is in Appendix C. This code was written for linux kernel 3.13.2, so you may have to modify it to fit your kernel. It may be helpful to look at other modules in /usr/src/linux-3.13.2/crypto, or in whatever directory your kernel source is installed. I will use 3.13.2 as my example. You need to copy the code from Appendix C into the file /usr/src/linux-3.13.2/crypto/rev16.c. You also need to add these lines to /usr/src/linux-3.13.2/crypto/Kconfig:

```
config CRYPTO_REV16
      tristate "cipher that reverses the order of each block of 16 bytes"
      select CRYPTO_ALGAPI
      help
        reverses the order of each block of 16 bytes
```

and this line into the appropriate place in /usr/src/linux-3.13.2/Makefile:

```
+obj-$(CONFIG_CRYPTO_REV16) += rev16.o
```

Then, run the configurator and select rev16 to be compiled as a module. For this, I usually use

```
cd /usr/src/linux-3.13.2
```

```
sudo make xconfig
```

Then build with

```
sudo make modules
```

Copy the new module into its proper place:

```
sudo cp crypto/rev16.ko /lib/modules/3.13.2/kernel/crypto/
```

and run depmod:

```
sudo depmod -a
```

Load the module into the kernel:

```
sudo modprobe rev16
```

When you are finished with this, return to the wd directory that you made at the start.

The JMS528S chip does everything backwards, so we need to reverse the bytes of our KEK:

```
cat kek.hex | grep -o .. | tac | echo "$(tr -d '\n')" > kek1.hex
```

Set up a decryption filter for the keyblock, sandwiching AES decryption between two layers of byte reversal:

```
echo "" | sudo cryptsetup -c rev16-ecb -d - create wd1 kb.bin
cat kek1.hex | xxd -p -r | sudo cryptsetup -c aes-ecb \
      --key-size=256 -d - --hash=plain create wd2 /dev/mapper/wd1
echo "" | sudo cryptsetup -c rev16-ecb -d - create wd3 /dev/mapper/wd2
```

To check that it worked, look for “DEK1” in the seventeenth line of the output of hexdump:

```
sudo hexdump -C /dev/mapper/wd3
```

```
000000f0  47 00 00 00 d2 00 00 00  3b 00 00 00 31 00 00 00  |G...Ò...;...1...|
00000100  44 45 4b 31 c1 91 00 00  59 e0 c8 57 3b af 60 55  |DEK1Á...YàÈW;¯`U|
00000110  cc 76 eb 00 e6 12 a3 92  03 1f 24 0a e8 10 ad e9  |Ìvë.æ.£...$.è.é|
```

Extract the DEK, which is in reverse order:

```
sudo dd if=/dev/mapper/wd3 bs=1 skip=268 count=16 of=dek0.bin
sudo dd if=/dev/mapper/wd3 bs=1 skip=288 count=16 of=temp
sudo chmod a+rw dek0.bin temp
cat temp >> dek0.bin
rm -f temp
```

Convert to hexadecimal and reverse to get the correct DEK:

```
xxd -p -c 32 dek0.bin > dek0.hex
cat dek0.hex | grep -o .. | tac | echo "$(tr -d '\n')" > dek.hex
```

Clean up by removing the encryption filter on the keyblock:

```
sudo cryptsetup remove wd3
sudo cryptsetup remove wd2
sudo cryptsetup remove wd1
```

Now that we have the DEK, we can mount the drive. In my example, the drive was at /dev/sdc, so I would use these commands:

```
echo "" | sudo cryptsetup -c rev16-ecb -d - create wd1 /dev/sdc
cat dek.hex | xxd -p -r | sudo cryptsetup -d - --hash=plain \
      --key-size=256 -c aes-ecb create wd2 /dev/mapper/wd1
echo "" | sudo cryptsetup -c rev16-ecb -d - create wd /dev/mapper/wd2
```

The backslash at the end of the second line indicates that the command continues on the next line.

Check for success:

```
sudo file -s /dev/mapper/wd
```

If you see something like

```
/dev/mapper/wd: DOS/MBR boot sector …
```

then you have succeeded in decrypting your disk.

Symwave SW6316 chip

Read the keyblock from the end of the disk. The location of this block depends on the size of your disk:

500 GB	976770435
750 GB	1465144707
1 TB	1953521027
2 TB	3907026307

So, for example, if you have a 2TB disk at sdc, use this command:

```
dd if=/dev/sdc bs=512 skip=3907026307 count=1 of=kb0.bin
```

Check to see that you have indeed obtained the keyblock by doing a hexdump and look for “WMYS”:

```
hexdump -C kb0.bin
```

```
00000000  57 4d 59 53 fa 01 01 f8  00 00 00 00 02 00 00 00  |WMYSú..ø........|
00000010  b7 1e 9a 37 36 40 5d db  42 25 89 a0 9e 97 b0 8d  |·..76@]ÛB%. ..°.|
00000020  fa bc 6f 46 4d 54 57 25  ab 44 02 e0 6a 5a 07 f0  |ú¼oFMTW%«D.àjZ.ð|
00000030  96 f4 79 ba e7 cc f2 80  15 88 b9 b5 72 b1 03 20  |.ôyºçÌò...¹µr±. |
00000040  f3 65 eb 88 91 70 f9 e7  09 9a ee cb 58 05 ad 97  |óeë..pùç..îËX..|
00000050  e3 6e b3 6d 5f 78 c9 cd  fe cb 85 c0 43 50 06 8d  |ãn³m_xÉÍþË.ÀCP..|
00000060  0f b6 50 6e 1a 36 30 8c  8e 25 9b fa 32 26 6b 6a  |.¶Pn.60..%.ú2&kj|
00000070  04 02 72 61 c0 a9 f3 65  a1 b4 b5 55 0c d4 e7 c7  |..raÀ©óe¡´µU.ÔçÇ|
00000080  f1 52 3b f2 46 b3 e8 69  00 00 00 00 00 00 00 00  |ñR;òF³èi........|
00000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000200
```

Convert the keyblock to hexadecimal:

```
xxd -p -c 16 kb0.bin > kb0.hex
```

The Symwave chip is based on a Motorola processor, so we have to fix the endianness of the keyblock. We do this by reversing the order of each 4-byte block with these three commands:

```
cat kb0.hex | grep -o ........ | tac | echo "$(tr -d '\n')" > temp
cat temp | grep -o .. | tac | echo "$(tr -d '\n')" | xxd -p -r > kb.bin
rm -f temp
```

Extract the wrapped disk encryption key (eDEK):

```
dd if=kb.bin bs=8 skip=2 count=5 of=edek.bin
```

We now need the unwrapper. The python code for it is in Appendix B. Copy it into a file called unwrap.py and make it executable:

```
chmod +x unwrap.py
```

Unwrap the DEK:

```
./unwrap.py `xxd -p -c 40 edek.bin` `cat kek.hex` > dek0.hex
```

We need to fix the endianness of the DEK:

```
cat dek0.hex | grep -o ........ | tac | echo "$(tr -d '\n')" > temp
cat temp | grep -o .. |tac | echo "$(tr -d '\n')" > dek.hex
rm -f temp
```

Now that we have the DEK, we can mount the drive. In my example, the drive was at /dev/sdc, so I would use this command:

```
cat dek.hex | xxd -p -r | sudo cryptsetup -d - --hash=plain \
     --key-size=256 -c aes-ecb create wd /dev/sdc
```

The backslash at the end of the first line indicates that the command continues on the next line.

Check for success:

```
sudo file -s /dev/mapper/wd
```

If you see something like

```
/dev/mapper/wd: DOS/MBR boot sector …
```

then you have succeeded in decrypting your disk.

Initio INIC-1607E chip

PLX OXUF943SE chip

Mounting

Next, depending on your system, the partitions might be mounted automatically. If not, then we must probe the partition table and load the results into the kernel. First, look in /dev for the last dm device:

```
ls /dev/dm-*
```

If you only have one mapped device, then it will be dm-0. If you have others, we want the one with the largest number. For example, take it to be dm-7. Add the partitions with

```
sudo kpartx -a /dev/dm-7
```

Your partitions will appear as /dev/mapper/wdp1 etc. Mount as follows:

```
sudo mkdir /mnt/wd1
sudo mount /dev/mapper/wdp1 /mnt/wd1
```

If nothing went wrong, then you can now access your files in /mnt/wd1.

Mounting can be automated at boot time, and the method for doing so varies from system to system.

Appendix A

Code for the bash script wd_kdf.sh:

```
#!/bin/bash

KEK=`echo -n "WDC.$1" | iconv -f UTF-8 -t UTF-16LE | xxd -p -c 64`

for i in `seq 1 1000`; do
    KEK=`echo -n $KEK | xxd -p -r | sha256sum | cut -d \  -f 1`
  done

echo $KEK
```

Appendix B

Code for the RFC 3394 unwrapping program unwrap.py, modified from https://gist.github.com/kurtbrose/4243633:

```
#!/usr/bin/python

import struct
from Crypto.Cipher import AES

QUAD = struct.Struct('>Q')

def aes_unwrap_key_and_iv(kek, wrapped):
    n = len(wrapped)/8 - 1
    R = [None]+[wrapped[i*8:i*8+8] for i in range(1, n+1)]
    A = QUAD.unpack(wrapped[:8])[0]
    decrypt = AES.new(kek).decrypt
    for j in range(5,-1,-1): #counting down
        for i in range(n, 0, -1): #(n, n-1, ..., 1)
            ciphertext = QUAD.pack(A^(n*j+i)) + R[i]
            B = decrypt(ciphertext)
            A = QUAD.unpack(B[:8])[0]
            R[i] = B[8:]
    return "".join(R[1:]), A

def aes_unwrap_key(kek, wrapped, iv=0xa6a6a6a6a6a6a6a6):
    key, key_iv = aes_unwrap_key_and_iv(kek, wrapped)
    if key_iv != iv:
        raise ValueError("Integrity Check Failed: "+hex(key_iv)+
                  " (expected "+hex(iv)+")")
    return key

if __name__ == "__main__":
    import sys
    import binascii
    CIPHER = binascii.unhexlify(sys.argv[1])
    KEK = binascii.unhexlify(sys.argv[2])
    print binascii.hexlify(aes_unwrap_key(KEK, CIPHER))
```

Appendix C

Code for the cryptographic module that simply reversed each block of 16 bytes. This was written for linux kernel 3.13.2. You may need to modify it to fit your system.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/crypto.h>
#include <asm/byteorder.h>

int rev16_setkey(struct crypto_tfm *tfm, const u8 *in_key,
        unsigned int key_len)
{
     return 0;
}

static void rev16_encrypt(struct crypto_tfm *tfm, u8 *out,
        const u8 *in)
{
     int i;
     u8 temp[16];
     for (i=0;i<16;i++)
          temp[i] = in[i];
     for (i=0;i<16;i++)
          out[i] = temp[15-i];
     return;
}

static struct crypto_alg rev16_alg = {
     .cra_name           =    "rev16",
     .cra_driver_name    =    "rev16",
     .cra_priority       =    100,
     .cra_flags          =    CRYPTO_ALG_TYPE_CIPHER,
     .cra_blocksize      =    16,
     .cra_ctxsize        =    0,
     .cra_alignmask      =    3,
     .cra_module         =    THIS_MODULE,
     .cra_u              =    {
          .cipher = {
               .cia_min_keysize    =    0,
               .cia_max_keysize    =    32,
               .cia_setkey         =    rev16_setkey,
               .cia_encrypt        =    rev16_encrypt,
               .cia_decrypt        =    rev16_encrypt
          }
     }
};
```

```
static int __init rev16_init(void)
{
     return crypto_register_alg(&rev16_alg);
}

static void __exit rev16_fini(void)
{
     crypto_unregister_alg(&rev16_alg);
}

module_init(rev16_init);
module_exit(rev16_fini);

MODULE_DESCRIPTION("reverses the bytes of each 16-byte block");
MODULE_LICENSE("GPL");
MODULE_ALIAS("rev16");
```

Acknowledgements

Information about these drives comes mainly from “got HW crypto” by G. Alendal, C. Kison, and modg (http://eprint.iacr.org/2015/1002.pdf) and from the source code of the reallymine project (https://github.com/andlabs/reallymine). The password unwrapping program in python is based on the one from Kurt Rose at https://gist.github.com/kurtbrose/4243633. Keyblock locations are from athomic1's information in the comments to the reallymine project.