Mouting encrypted WD drives in linux

Thomas Kaeding (thomas.a.kaeding@gmail.com)
20170205; last modified 20170509 as version 0.3

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.

This document contains instructions only for drives that use these chips for hardware encryption:

JMicron JMS538S
Symwave SW6316
Initio INIC-1607E
PLX OXUF943SE

We also only deal with 256-bit keys and AES encryption in ECB mode. There are a few drives with 128-bit keys or XTS mode. Dealing with them is still work in progress.

You will need the following software packages. Be aware that the exact names can vary among linux distributions. Most of these, except kernel development packages, are included with most installations of linux.

bash
GNU coreutils
util-linux
sudo
cryptsetup
openssl
vim (for the xxd utility)
file
multipart-tools

For the Symwave chip, you also need

python
pycrypto

For the JMicron and Initio chips, you also need

kernel development (may be called “kernel-dev”)
    (in Ubuntu, use “sudo apt-get build-dep linux-image-$(uname -r)”)
gcc (same version that was used to build your kernel)
kmod or modutils

First Steps

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 JMS538S chip

Read the keyblock from the end of the disk. The location of this block depends on the size of your disk. The middle column in this table is decimal, the third column hexadecimal.

50 0GB	976769056	0x03A385020
750 GB	1465143328	0x057545020
1 TB	1953519648	0x074705820
2 TB	3907024928	0x0E8E07820
3 TB	5860528160	0x15D509020
4 TB	7814031392	0x1D1C0A820

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 kb.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..¶|
```

If the keyblock was not found, try the script in Appendix D.

The JMS538S 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
```

In order to extract the disk encryption key (DEK), we have to reverse each block of 16 bytes, decrypt with AES in ECB mode, and then reverse each block again. These three commands will do it:

```
for i in `seq 0 31`; do
     dd if=kb.bin bs=16 count=1 skip=$i status=none | \
          xxd -p | grep -o .. | tac | echo "$(tr -d '\n')" | \
          xxd -p -r >> kb1.bin
     done
openssl enc -d -aes-256-ecb -K `cat kek1.hex` \
     -nopad -in kb1.bin -out kb2.bin
for i in `seq 0 31`; do
     dd if=kb2.bin bs=16 count=1 skip=$i status=none | \
          xxd -p | grep -o .. | tac | echo "$(tr -d '\n')" | \
          xxd -p -r >> kb3.bin
     done
```

The backslashes at the ends of lines indicate that the command is continued on the next line.

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

```
sudo hexdump -C kb3.bin
```

```
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 disk encryption key (DEK), which is in reverse order:

```
dd if=kb3.bin bs=1 skip=268 count=16 of=dek0.bin status=none
dd if=kb3.bin bs=1 skip=288 count=16 status=none >> dek0.bin
```

Convert to hexadecimal and reverse it to get the correct DEK:

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

Now for the hardest part: we need to build a new encryption module for the kernel. We will use this module merely to reverse 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 and instructions for building it are in Appendix C. When you are finished building it, return to the wd directory that you made at the start.

Now that we have the DEK and the new module, we can set up the encryption filter. In my example, the drive was at /dev/sdc, so I would use these commands:

```
echo "fedcba9876543210" | xxd -p -r | sudo cryptsetup \
     -d - --hash=plain -c permute-ecb --key-size=64 \
     create wd-layer1 /dev/sdc
cat dek.hex | xxd -p -r | sudo cryptsetup -d - --hash=plain \
     --key-size=256 -c aes-ecb create wd-layer2 /dev/mapper/wd-layer1
echo "fedcba9876543210" | xxd -p -r | sudo cryptsetup \
     -d - --hash=plain -c permute-ecb --key-size=64 \
     create wd /dev/mapper/wd-layer2
```

Check for success:

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

If you see something like

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

then you have succeeded in decrypting your disk.

You can delete all the temporary files that we created, *except* `dek.hex` and `permute.ko`.You will need them to mount the disk again in the future.

Symwave SW6316 chip

Read the keyblock from the end of the disk. The location of this block depends on the size of your disk. The middle column in this table is decimal, the third column hexadecimal.

500 GB	976770435	0x03A385583
750 GB	1465144707	0x057545583
1 TB	1953521027	0x074705D83
2 TB	3907026307	0x0E8E07D83

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........|
```

If the keyblock was not found, try the script in Appendix D.

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 this command:

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

The backslash at the end of the first line means that the command is continued on the next line.

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. First, be sure that the pycrypto package is already installed on your system. The python code for the unwrapper is in Appendix B. Copy it into a file called `unwrap.py` and make it executable:

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

Unwrap the disk encryption key (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 "$(tr -d '\n')" | \
        grep -o .. |tac | echo "$(tr -d '\n')" > dek.hex
```

Now that we have the DEK, we can set up the encryption filter. 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
```

Check for success:

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

If you see something like

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

then you have succeeded in decrypting your disk.

You can delete all the temporary files that we created, *except* `dek.hex`. You will need it to mount the disk again in the future.

Initio INIC-1607E chip

Read the keyblock from the end of the disk. The location of this block depends on the size of your disk. The middle column in this table is decimal, the third column hexadecimal.

500 GB	976769032	0x03A385008	
750 GB	1465143304	0x057545008	
1 TB	1953519624	0x074705808	(unverified)
2 TB	3907024904	0x0E8E07808	

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

```
dd if=/dev/sdc bs=512 skip=3907024904 count=1 of=kb.bin
```

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

```
hexdump -C kb.bin

00000000  57 44 01 14 00 00 00 00  00 00 00 00 00 00 00 00  |WD..............|
00000010  00 00 00 00 1d 07 68 00  00 00 00 00 1d 07 68 00  |......h.......h.|
00000020  00 00 00 00 00 14 e0 00  20 00 00 00 00 00 00 00  |......à. .......|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 57 44 01 14  |............WD..|
00000040  32 92 ed 81 13 26 9e 98  df 1b a4 87 ef c6 37 3c  |2.í..&..ß.¤.ïÆ7<|
```

If the keyblock was not found, try the script in Appendix D.

We have to fix the endianness of the keyblock. We do this by reversing the order of each 4-byte block with this command:

```
cat kb.bin | xxd -p -c 32 | grep -o ........ | tac | \
     echo "$(tr -d '\n')" | grep -o .. | tac | \
     echo "$(tr -d '\n')" | xxd -p -r > kb1.bin
```

The backslash at the end of the first line means that the command is continued on the next line.

The KEK also has to be fixed. We must swap its two halves, and reverse it. We can do this all in one fell swoop with this command:

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

Decrypt the keyblock:

```
openssl enc -d -aes-256-ecb -K `cat kek1.hex` \
     -nopad -in kb1.bin -out kb2.bin
```

Extract the disk encryption key (DEK):

```
dd if=kb2.bin bs=4 skip=103 count=8 | xxd -p -c 32 > dek1.hex
```

We have to fix up this DEK by reversing each half and changing the endianness:

```
cat dek1.hex | grep -o ................................ | tac | \
     echo "$(tr -d '\n')" | grep -o ........ | tac | \
     echo "$(tr -d '\n')" > dek.hex
```

Now for the hardest part: we need to build a new encryption module for the kernel. We will use this module merely to reverse the order of each 4-byte block. You will need to install all the necessary packages for kernel development. The C code for the new module and instructions for building it are in Appendix C. When you are finished building it, return to the wd directory that you made at the start.

Now that we have the DEK and the new module, we can set up the encryption filter. In my example, the drive was at /dev/sdc, so I would use these commands:

```
echo "32107654ba98fedc" | xxd -p -r | sudo cryptsetup \
     -d - --hash=plain -c permute-ecb --key-size=64 \
     create wd-layer1 /dev/sdc
cat dek.hex | xxd -p -r | sudo cryptsetup -d - --hash=plain \
     --key-size=256 -c aes-ecb create wd-layer2 /dev/mapper/wd-layer1
echo "32107654ba98fedc" | xxd -p -r | sudo cryptsetup \
     -d - --hash=plain -c permute-ecb --key-size=64 \
     create wd /dev/mapper/wd-layer2
```

Check for success:

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

If you see something like

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

then you have succeeded in decrypting your disk.

You can delete all the temporary files that we created, *except* `dek.hex` and `permute.ko`. You will need them to mount the disk again in the future.

PLX OXUF943SE chip

Read the keyblock from the end of the disk. The location of this block depends on the size of your disk. The middle column in this table is decimal, the third column hexadecimal.

500 GB	?	?
750 GB	?	?
1 TB	?	?
2 TB	?	?
3 TB	5860533120	15D50A380

So, for example, if you have a 3TB disk at sdc, use this command:

```
dd if=/dev/sdc bs=512 skip=5860533120 count=1 of=kb.bin
```

Check to see that you have indeed obtained the keyblock by doing a hexdump and look for "SInE":

```
hexdump -C kb.bin
```

```
00000000  53 49 6e 45 01 00 00 00  04 00 64 01 01 85 84 00  |SInE......d.....|
00000010  01 00 00 00 dc 22 c2 ed  f2 a5 7f 73 23 cf 58 28  |....Ü"Âíò¥.s#ÏX(|
00000020  4d 6f 4d 6f b5 fb 1a d1  9f f9 2a 72 70 51 93 b8  |MoMoµû.Ñ.ù*rpQ.¸|
00000030  4b 74 2c 6a 67 19 3f 4c  c1 f9 57 6f ab e6 07 e5  |Kt,jg.?LÁùWo«æ.å|
00000040  db e0 49 3c ad 00 89 b3  0d cb ef a1 e7 c5 75 9a  |ÛàI<-..³.Ëï¡çÅu.|
00000050  e2 db 1f 5f ff ff ff ff  ff ff ff ff ff ff ff ff  |âÛ._ÿÿÿÿÿÿÿÿÿÿÿÿ|
```

If the keyblock was not found, try the script in Appendix D.

In order to decrypt the keyblock, we need to remove 20 bytes from the beginning:

```
dd if=kb.bin bs=4 skip=5 count=64 of=kb1.bin
```

We also need to reverse the order and fix the endianness of the KEK:

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

Decrypt the keyblock:

```
openssl enc -d -aes-256-ecb -K `cat kek1.hex` \
     -nopad -in kb1.bin -out kb2.bin
```

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

Extract the disk encryption key (DEK):

```
dd if=kb2.bin bs=32 count=1 | xxd -p -c 32 > dek1.hex
```

Reverse the order and fix the endianness of the DEK:

```
cat dek1.hex | grep -o ........ | tac | echo "$(tr -d '\n')" > dek.hex
```

Now that we have the DEK, we can set up the encryption filter on the disk. 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
```

Check for success:

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

If you see something like

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

then you have succeeded in decrypting your disk.

You can delete all the temporary files that we created, *except* `dek.hex`. You will need that file to mount the disk again in the future.

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. Add the partitions with

```
sudo kpartx -a /dev/mapper/wd
```

Your partitions will appear as `/dev/mapper/wd1` etc. Mount as follows:

```
sudo mkdir /mnt/wd1
sudo mount /dev/mapper/wd1 /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. It requires that the pycrypto package be installed.

```
#!/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 permutes each block of 16 bytes. This was written for linux kernel 3.13.2. You may need to modify it to fit your system. Instructions for building are below.

```
#include <linux/module.h>
#include <linux/crypto.h>

static u8 encvec[16], decvec[16];

int permute_setkey(struct crypto_tfm *tfm, const u8 *in_key,
           unsigned int key_len)
{
        int i;
        for (i=0;i<key_len;i++) {
                encvec[2*i]   = in_key[i] >> 4;
                encvec[2*i+1] = in_key[i] & 0x0f;
        }
        for (i=0;i<16;i++)
                decvec[encvec[i]] = (u8)i;
        return 0;
}

static void permute_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[encvec[i]];
        return;
}

static void permute_decrypt(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[decvec[i]];
        return;
}

static struct crypto_alg permute_alg = {
        .cra_name               =       "permute",
        .cra_driver_name        =       "permute",
        .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        =       8,
                        .cia_max_keysize        =       8,
                        .cia_setkey             =       permute_setkey,
                        .cia_encrypt            =       permute_encrypt,
                        .cia_decrypt            =       permute_decrypt
                }
        }
};

static int __init permute_init(void)
{
        return crypto_register_alg(&permute_alg);
}

static void __exit permute_fini(void)
{
        crypto_unregister_alg(&permute_alg);
}

module_init(permute_init);
module_exit(permute_fini);

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

In order to build this module, you will need to install all the necessary packages for kernel development. 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.

Copy the code for the module into a new file call it `permute.c`.

Create a makefile:

```
echo “obj-m := permute.o” > Makefile
```

Build:

```
make -C /lib/modules/`uname -r`/build M=$PWD
```

This creates a file called `permute.ko`. Load the module into the kernel:

```
sudo insmod permute.ko
```

If you want to have the module permanently on your system, copy it:

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

and configure it:

```
sudo depmod
```

After that, anytime you want to load it, use this command:

```
sudo modprobe permute
```

Appendix D

Bash script to help find the keyblock at the end of a disk, after it has been removed from the original enclosure. This script is called with one argument, which is the name of the drive in linux, such as “sdc”, without the preceeding “/dev/”.

```
#!/bin/bash

DEVICE="$1"

SIZE=`cat /proc/partitions | grep $DEVICE | awk '{print $3}'`
SIZE=`expr $SIZE \* 2`

LOWERLIMIT=`expr $SIZE - 8192` # 4 MB should be enough

for i in `seq $SIZE -1 $LOWERLIMIT`; do
    FIRSTLINE=`dd if=/dev/$DEVICE skip=$i count=1 status=none | \
          xxd -p | head -n 1`
    if [ `echo $FIRSTLINE | grep "^57447631"` ]; then
        echo "found JMicron keyblock at sector $i"
        break
      fi
    if [ `echo $FIRSTLINE | grep "^574d5953"` ]; then
        echo "found Symwave keyblock at sector $i"
        break
      fi
    if [ `echo $FIRSTLINE | grep "^57440114"` ]; then
        echo "found Initio keyblock at sector $i"
        break
      fi
    if [ `echo $FIRSTLINE | grep "^53496e45"` ]; then
        echo "found PLX keyblock at sector $i"
        break
      fi
  done
echo "dumping to kb.bin"
dd if=/dev/$DEVICE skip=$i count=1 of=kb.bin status=none
```

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). Much of the same information can be found in 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. Most of the keyblock locations are from athomic1's information in the comments to the reallymine project.