PV204: Disk encryption lab

April 17, 2024, Milan Broz <xbroz@fi.muni.cz>

Introduction

Encryption can provide confidentiality of user data. It can be implemented on different layers, including applications, file systems, or storage devices.

Application layer encryption examples are PGP or ZIP compression with a password.

Encryption of files (inside file system or through independent layer like Linux fscrypt) provides a more generic solution. However, this solution provides encrypted data with a key per user. Yet some parts (like file system metadata) are not encrypted.

The low-level storage (disk) encryption is called *Full Disk Encryption* (FDE). It is entirely transparent for the user (no need to choose what to encrypt – the whole disk is encrypted). The encrypted disk behaves the same as a disk without encryption.

The primary use of FDE is to provide data confidentiality in power-down mode (stolen laptop does not leak user data). The major disadvantage is that everyone who knows the password can read the whole disk. Often we combine FDE with another encryption layer.

Once the disk is unlocked, the media encryption key remains in the system, usually directly in system RAM. Exercise II will show how easy it is to get this key from the system's memory image.

Another disadvantage of FDE is that it usually cannot guarantee data integrity. Encryption is fully transparent and length-preserving; the ciphertext and plaintext have the same size. There is no space to store any integrity information. This allows attacks by direct modification of ciphertext.

The FDE works on the sector level, the same as the block device. Atomic I/O access units for encrypted devices are sectors. Sectors are encrypted independently by the symmetric cipher. Cipher block size is (in most cases) smaller than the size of a sector. It means that inside the sector, we need to use an encryption mode (suitable for storage encryption).

It is crucial that the same data in different sectors must produce different ciphertext. This is achieved by a per-sector tweak (IV, Initialization Vector) that is usually derived from logical sector offset (sector number). A combination of properly used IV and encryption mode is critical for the security of the FDE system. *Figure 1* shows an example of a wrongly used mode for encryption (here ECB, Electronic Codebook, and XTS with an incorrect constant tweak). Another problem could appear with CBC mode [CBC] and predictable IV. Then the attacker can use the predictability of IV and create watermarks and detect them directly from the ciphertext. We will show this flaw on the legacy TrueCrypt device in Exercise III (an optional exercise).

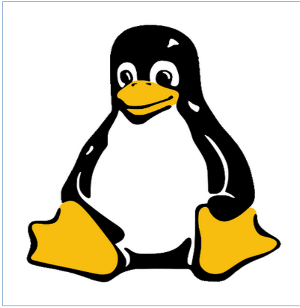
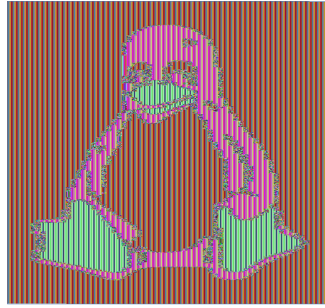
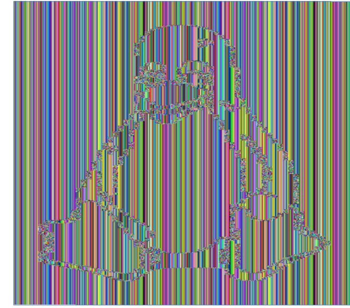
  

Figure 1. An image encrypted in ECB mode and XTS mode without tweaked-sector encryption.

Today, the most used encryption mode for disks is XTS [XTS]. This mode is constructed in such a way that it can safely use predictable sector numbers directly as an IV. The schema of CBC and XTS encryption mode is for reference in Appendix.

FDE can be implemented on the hardware layer (self-encrypted disk, encryption inside controller chipset) or in software [FDE]. Software implementation is, for example, BitLocker for Windows systems, dm-crypt in Linux, FileVault on macOS, or VeraCrypt. There are many other tools implementing this functionality [COMP].

In the following exercises, we will use two software FDE tools. The first one is LUKS/dm-crypt [LUKS], which is the primary solution for Linux-based systems and VeraCrypt [VC].

Both tools can create a virtual encrypted block device within a partition or a regular file.
This virtual disk can be formatted as if it was a physical device and mounted.
VeraCrypt also supports trivial steganography and allows to create a hidden virtual volume within another volume.

References

[FDE] Disk encryption, https://en.wikipedia.org/wiki/Disk_encryption,
[COMP] Comparison of disk encryption software, https://en.wikipedia.org/wiki/Comparison_of_disk_encryption_software
[CBC] Cipher Block Chaining mode, https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
[XTS] XTS-AES Mode for Confidentiality on Storage Devices, https://csrc.nist.gov/publications/detail/sp/800-38e/final
[LUKS] Linux Unified Key Setup / cryptsetup, https://gitlab.com/cryptsetup/cryptsetup/
[VC] VeraCrypt (former TrueCrypt), Free open-source disk encryption software, https://www.veracrypt.fr

Exercises

We will use prepared Virtual Machine PV204 (slightly modified Debian Linux).
virtual machine can be run in Oracle VirtualBox.
The virtual machine has preinstalled LUKS disk encryption of the whole system, **cryptsetup** 2.x (it can open TrueCrypt,VeraCrypt, BitLocker and FileVault2 devices as well), **TrueCrypt** 7.1a and **VeraCrypt** 1.24.

> **User name: pv204**
> **Password** (also unlocking password for disk)**: pv204**

Note: User pv204 can run superuser commands via sudo without need of entering password, e.g. ***sudo ls /root****.*

Exercise: Obtaining encryption key from VM memory image

For most of the software-based disk encryption applications, the encryption runs on the CPU, usually inside the OS context (optionally with HW acceleration like AES-NI instructions). In this operating mode, the encryption key is present in RAM during the operation. An attacker can obtain an image of physical memory and try to search for the encryption key. This image can be either obtained through software (we will use VirtualBox debugger function) or through hardware (Cold boot attack, i.e., recover memory content after force reboot/short powerdown or use FireWire hw debug functions). The obtained key can be directly used for storage decryption (without password).

The next problem is to identify candidate keys in memory image. This can be done by recognizing internal OS structures in memory or by a more generic approach, like reconstructing AES keys by identifying specific round keys [COLD].

Your Task

1. Copy provided image and prepare Virtual Machine with some active and mounted encrypted devices and understand the storage stack inside.

   You should have two encrypted and active mappings for this exercise in VM. One is system encryption itself (LUKS/dm-crypt) and second will be VeraCrypt device created from disk image **testdisk.img**.

   a) Start and login to VM.
   b) Run VeraCrypt GUI (on desktop) and select image **testdisk.img** (please use AES only for encryption algorithm). If you want to create a new image, be sure to enter an absolute path (like /home/pv204/name.img) otherwise VeraCrypt format does not work properly.
   c) Mount image through VeraCrypt GUI and investigate used parameters (Volume Properties). Keep it mounted to investigate key in memory later.
   d) Investigate and understand storage stack, dump parameters about encrypted virtual devices. Use **lsblk** command to get the whole picture:

```
root@pv204:/home/pv204# lsblk
NAME                     MAJ:MIN RM  SIZE RO TYPE  MOUNTPOINT
loop0                      7:0    0   32M  0 loop
└─veracrypt1             254:3    0 31.8M  0 dm    /media/veracrypt1
sda                        8:0    0   16G  0 disk
├─sda1                     8:1    0  243M  0 part  /boot
├─sda2                     8:2    0    1K  0 part
└─sda5                     8:5    0 15.8G  0 part
  └─sda5_crypt           254:0    0 15.8G  0 crypt
    ├─pv204--vg-root     254:1    0 15.1G  0 lvm   /
    └─pv204--vg-swap_1   254:2    0  708M  0 lvm   [SWAP]
```

   In this example, **sda5** is your system encrypted partition, **loop0** is the VeraCrypt (or TrueCrypt) device mapped to **testimage.img**.

2. Check VeraCrypt device - note the header is encrypted so you cannot get information using **blkid**. Use cryptsetup TrueCrypt/VeraCrypt extension to get more info about device on-disk header:

```
$ blkid testdisk.img
$ cryptsetup tcryptDump testdisk.img --dump-master-key
```

```
Enter passphrase for testdisk.img:
TCRYPT header information for testdisk.img
Cipher chain:   aes
Cipher mode:    xts-plain64
Payload offset: 256
MK bits:        512
MK dump:        a8 8e 10 08 2d a1 a1 c3 b3 bf f0 4d 61 ab 16 1b
                80 3d 4a 17 ce 56 eb 02 a5 df 72 41 55 b5 4b 9c
                bb d0 19 94 01 be 3b 97 11 6f e7 04 29 ba bb c4
                19 17 8f 2e 7b 30 f9 f3 8c 10 41 35 6a 89 0d 91
```

Note: for older systems you have to add `--veracrypt` option to cryptsetup option.

Now, try to display encrypted keys directly from kernel (superuser can display full device-mapper dm-crypt mapping table for active devices).

```
pv204@pv204:~$ sudo dmsetup table --showkeys veracrypt1
0 65024 crypt aes-xts-plain64 a88e10082da1a1c3b3bff04d61ab161b803d4a17ce56eb02a5df724155b54b9cbbd0199
401be3b97116fe70429babbc419178f2e7b30f9f38c1041356a890d91 256 7:0 256
```

e) Repeat the same exercise for LUKS device (**/dev/sda5** mapped to **sda5_crypt**), use **`luksDump`** command and use **sudo** as access to the device node requires root. Note LUKS has visible header on-disk.

3. While the VM is still running, obtain snapshot of memory core image.
   You can use **Vbox_save_memcore.bat** script which uses internal VirtualBox debugger to dump VM memory core (it is in fact ELF format dump, but it is enough for our use).

4. Analyze the memory core image with the provided aeskeyfind [AESKEYFIND] program. Note this program search for specific AES structures. (In reality you have key in memory more times but in different format, just think about screen buffer with text dumps executed in step 1.)

5. Compare possible encryption keys with information you obtained in step 1.
   See XTS encryption mode definition [XTS] and think why you see two separate AES keys for one device.

6. Now, try to answer some questions:

   What can be the other keys in the dump?
   Why some keys repeats in memory dump?
   The XTS keys are swapped, why?
   Why VeraCrypt does not encrypt keys in memory (as in Windows)? Does it help?

7. Optionally, try to dump other types of encrypted devices using cryptsetup:
   LUKS - luksDump, BitLocker - bitlkDump, FileVault2 - fvault2Dump

References

[COLD] *Lest We Remember*: Cold Boot Attacks on Encryption Keys
https://citp.princeton.edu/research/memory/
[AESKEYFIND] Fixed and extended AESkeyfind source code (gcc optimization bug patch)
https://github.com/mbroz/aeskeyfind

Exercise: Data integrity protection with veritysetup and integritysetup

The previous exercise used kernel *dm-crypt* driver as an encryption engine for sector-level encryption. Another useful device-mapper target is *dm-verity*, which provides cryptographically strong sector-level data integrity protection for read-only devices, and *dm-integrity*, which utilizes a data integrity field (DIF) for storing additional data per sector.
The *dm-verity* driver uses an existing block device with data to protect and calculate the *Merkle tree* (tree of hashes) for verification of data (and hash tree). Hashes are stored separately (in a separate area or device). Optionally, there can be another device or area with redundant *forward-error correction* (FEC) data that can repair a block device data or hash tree corruption. Configuration is done through the *veritysetup* tool (part of the cryptsetup project). The *dm-verity* was initially developed as part of Android verified boot.
The *dm-integrity* driver provides additional per-sector space for writable devices and can also implement data integrity checking with various algorithms. Combined with the *dm-crypt* above (supported by LUKS2 format), it can utilize true authenticated encryption at the sector level. To configure *dm-integrity* devices, you can use *integritysetup* for standalone integrity devices or *cryptsetup* for the LUKS2 option. Note that *dm-integrity* does not try to solve replay attacks (replacement of old but valid data).

Your task

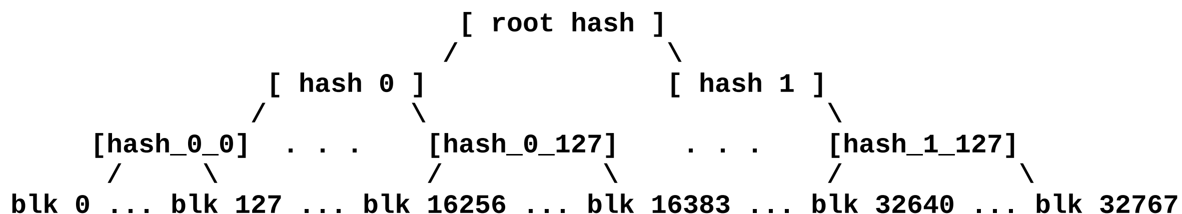
1. Format the existing image of the block device with veritysetup.

Create some test container with random data (or you can use a real LUKS image).
*# **dd if=/dev/urandom of=data.img bs=4k count=32768***

Format it with veritysetup
*# **veritysetup format data.img hash.img***

```
VERITY header information for hash.img
UUID:                   <random UUID>
Hash type:              1
Data blocks:            32768
Data block size:        4096
Hash blocks:            259
Hash block size:        4096
Hash algorithm:         sha256
Salt:                   <random salt in hexa>
Root hash:              <root hash in hexa>
Hash device size:       1064960 [bytes]
```

The command above creates an image with a stored Merkle tree and a separate root hash.
| data.img | hash.img |
You need to remember the root hash (in real systems, it is stored and signed by a trusted key).
You can visualize the content of the hash.img like this:
Merkle tree, alg = sha256, num_blocks = 32768, block_size = 4096



2. Verify block device integrity in userspace with veritysetup and in the kernel.

veritysetup verify data.img hash.img <root hash hexa> --verbose

```
Command successful.
```

sha256sum data.img

```
<sha256 sum>
```

sudo veritysetup open data.img test hash.img <root hash hexa>
sudo veritysetup status test
...

sudo sha256sum /dev/mapper/test

```
<sha256 sum>
```

Be sure to close the veritysetup device as we will format it again in the next step.
sudo veritysetup close test

3. Format the same image but also add a forward error correction image.

veritysetup format data.img hash.img --fec-device fec.img

```
VERITY header information for hash.img
UUID:                   <random UUID>
Hash type:              1
Data blocks:            32768
Data block size:        4096
Hash blocks:            259
Hash block size:        4096
Hash algorithm:         sha256
FEC RS roots:           2
FEC blocks:             262
Salt:                   <random salt in hexa>
Root hash:              <root hash in hexa>
Hash device size:       1064960 [bytes]
FEC device size:        1073152 [bytes]
```

This will create images with additional hash.img (Merkle tree) and fec.img (RS code).

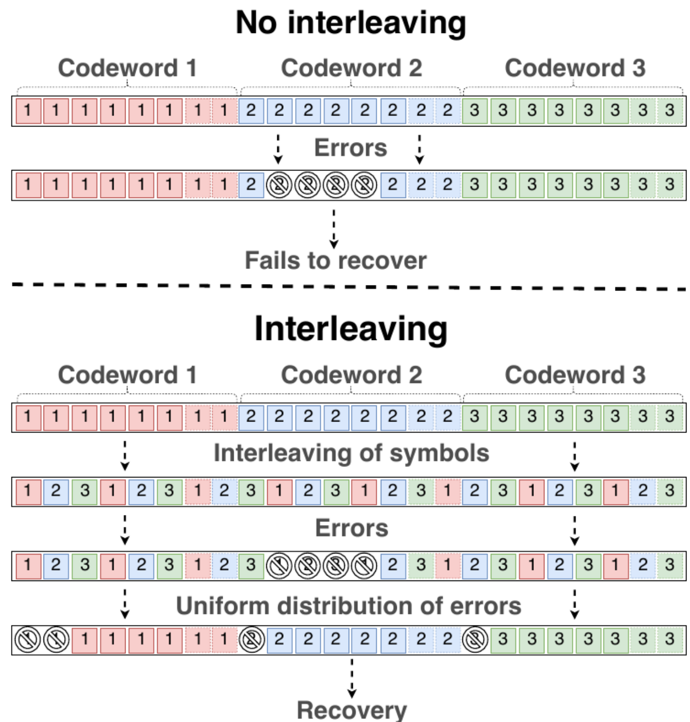
| data.img | hash.img | fec.img |

The forward error correction (FEC) in dm-verity in non-cryptographic operation based on block error correction code, here Reed-Solomon code, defined as *RS(M,N)*, that should be able to correct *(M-N)/2* symbol errors and *M-N* erasures.

The dm-verity uses fixed M = 255 (bytes) and symbol size 8-bit (byte). Veritysetup commandline uses a number of roots R = M - N as an option. For example, with R = 2 (default), we have RS(255, 253), symbol is byte, and codeword looks like **| data bytes (253) | parity bytes (2) |.**

Because we operate with fixed sectors (here 4096 bytes), all data, hashes and FEC data must be aligned to blocks (this defines some minimal image size and data alignment).

Note that redundant data codewords (stored in fec.img) are not stored consecutively, but with fixed interleaving. Why interleaving? Imagine what happens for error recovery of a larger erasure with and without interleaving:



4. Verify the block device in userspace (repeat step 2).

veritysetup verify data.img hash.img <root hash hexa> --fec-device fec.img --verbose

```
Command successful.
```

5. Introduce data corruption to the image and repeat verification.
You can use hexedit editor (man hexedit; F2 - save file, ctrl+c exit).

veritysetup verify data.img hash.img <root hash hexa> --fec-device fec.img --verbose

```
Verification failed at position 134213632.
Verification of data area failed.
Found 4 repairable errors with FEC device.
Command successful.
```

sudo veritysetup open data.img test hash.img <root hash hexa> --fec-device fec.img
sudo veritysetup status test
...
sudo sha256sum /dev/mapper/test

```
<sha256 sum>
```

sudo dmesg | grep verity-fec

```
 device-mapper: verity-fec: 7:1: FEC 69632: corrected 4 errors
```

sudo veritysetup close test

6. Now, try to answer some questions:
- *What happens if an error is detected but cannot be corrected?*
- *What happens if an error is present but neither detected nor corrected? Could this even happen?*
- *What happens if the root hash is incorrect? Can you recover from this situation?*

7. Now, try to use dm-integrity combined with dm-crypt for authenticated encryption.

First, create some empty container (here just sparse file):
\# ***truncate -s 128M aead.img***

AEAD (authenticated encryption with additional data) can be constructed either as length-preserving encryption (like AES-XTS) with an additional cryptographically strong algorithm for authentication tag (for example, HMAC) or directly with the authenticated encryption algorithm (like Chacha20-poly1305 or AEGIS). As the Linux kernel supports many combinations, here are two examples:

Add HMAC integrity protection to default cipher (aes-xts-plain64):
\# ***sudo cryptsetup luksFormat aead.img --integrity hmac-sha256***
\# ***cryptsetup luksDump aead.img***
```
...
      cipher: aes-xts-plain64
      sector: 4096 [bytes]
      integrity: hmac(sha256)
...
```

Or use a real AEAD algorithm, here AEGIS with 128bit key and random IV:
 # sudo cryptsetup luksFormat aead.img --integrity aead --cipher aegis128-random --key-size 128
```
...
      cipher: aegis128-random
      sector: 4096 [bytes]
      integrity: aead
...
```

\# ***sudo cryptsetup open aead.img test***

\# ***lsblk***
```
NAME       MAJ:MIN SIZE TYPE
loop0        7:0   128M loop      <-- loop for aead.img
└─test_dif 254:0 110.2M crypt     <-- dm-integrity, note space for DIF (auth.tags)
 └─test    254:1 110.2M crypt     <-- dm-crypt device
```

\# ***sudo cryptsetup close test***

You can repeat the same data corruption test as for dm-verity.
(Better reopen the device after introducing corruption to flush caches.)
Check kernel log - data corruption should be reported:

\# ***sudo sha256sum /dev/mapper/test***
```
sha256sum: /dev/mapper/test: Input/output error
```

\# ***sudo dmesg |grep crypt:***
```
device-mapper: crypt: dm-0: INTEGRITY AEAD ERROR, sector 225696
Buffer I/O error on dev dm-1, logical block 28212, async page read
```

Exercise: Hardware keylogger use

"If you let your machine out of your sight, it's no longer your machine."

In the context of disk encryption, a specialized device can be used to reveal entered passphrases (BIOS password, disk encryption unlocking passphrase).

A hardware keylogger is a device that intercepts data entered through the keyboard and stores it internally for later analysis. The device can also transmit data using a wireless network or another hidden channel.

If properly masked (e.g., in keyboard controller), it is very hard to reveal it by the user because the device is completely transparent to BIOS and operating system (and power consumption is minimal).

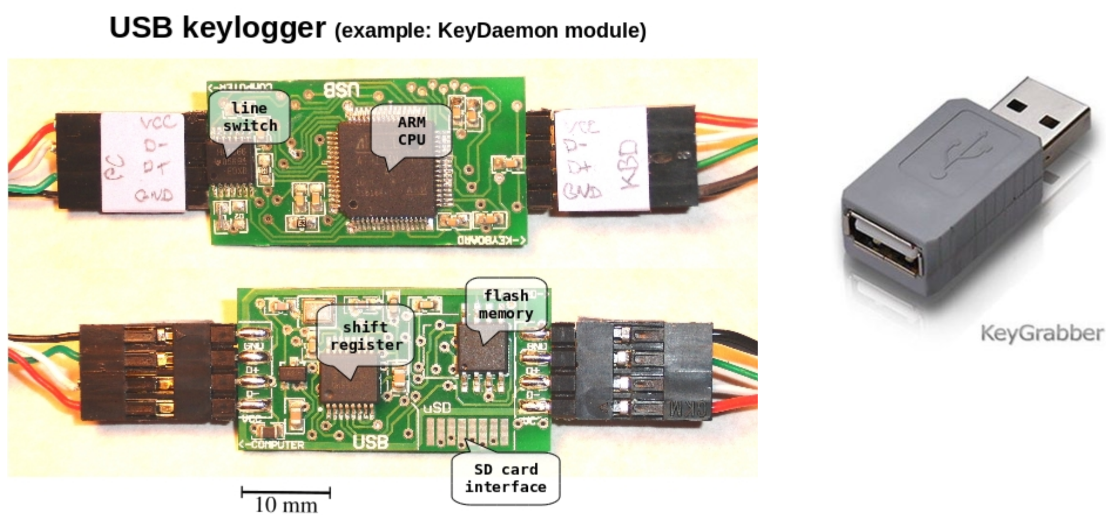


Figure 1. Example keylogger module

We will use a simple example of a commercially available USB keylogger module. The older version provides simulated flash-drive access for data retrieval; the new one can be accessed through a WiFi network (the key module is the WiFi access point in our example).

There is a limited amount of memory for the internal log, but the extended version provides an SD card interface for the log store (enough storage for almost the whole physical life of the keyboard).

Log retrieval is activated by an activation shortcut, which switches the keylogger into USB drive emulation mode and provides direct access to log storage (file). (It can also send UDP packets to process data in an external application.) the new version can be monitored in real-time through the internal webserver accessed through WiFi.

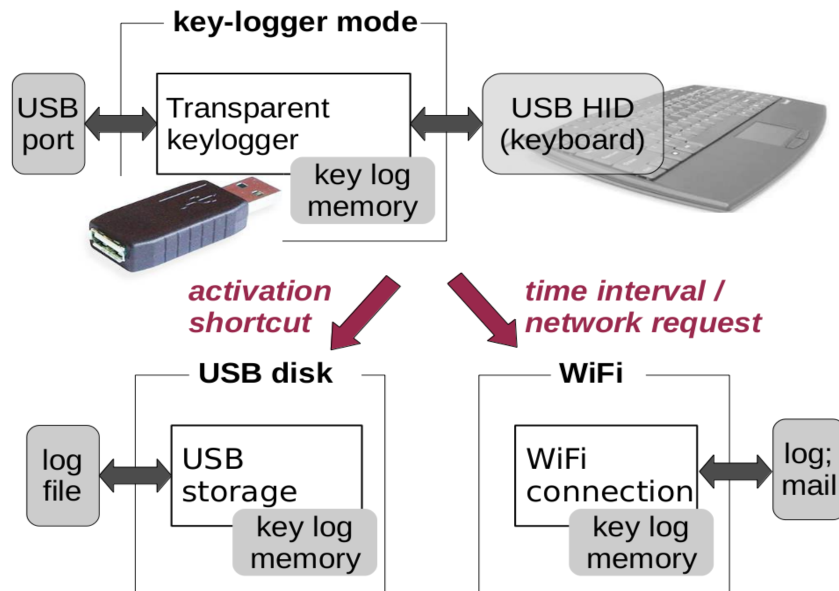


Figure 2. Log retrieval from the module

Your task

1. Connect the provided USB keyboard with keylogger module to USB port.
2. Check that the system recognizes keyboard (as the same as it is connected directly).
3. Simulate some work where you need enter sensitive data, e.g. create new TrueCrypt encrypted device, boot testing virtual machine and log in or try to enter some password in browser.
4. **For older, flash-drive keylogger** (see description on sticker)
    1. Switch to host system file explorer and press activation shortcut :
       'K' + 'B' + 'S' key together.
       The keyboard should detach and you should see new virtual USB flash disk.
    2. Investigate "LOG.txt" file on the disk and compare it with your work in step 3.
    3. If you want, delete the "LOG.txt" so your colleagues or lecturer will not see your "sensitive" data :-)
5. **For new, WiFi capable keyloggers**
    1. Keep device connected in USB port
    2. Connect your device (phone or laptop) to WiFi network KLOG1 or KLOG2 (see sticker on keylogger). NOTE: the network is open, (intentionally) not encrypted!
    3. Open browser at http://192.168.4.1 (for recent browser you will need to enforce non-https connection
    4. Investigate the log, you can still type on connected keyboard; refresh the page.
    5. If you want, erase the log in setting page so your colleagues or lecturer will not see your "sensitive" data :-)

    Please, DO NOT CHANGE WIFI SETTINGS or ACTIVATION KEYS!

Optional Exercise: Revealing TrueCrypt hidden disk existence (CBC)

The goal of this exercise is to show that old CBC mode in TrueCrypt is vulnerable to watermarking attack and that such attack could be used to reveal hidden disk existence just from ciphertext analysis.

Theory

The TrueCrypt (in version before 4.1) used CBC mode with Initialization Vector calculated from the sector number and xored with secret key K_{IV}. Also there is additional whitening which is calculated using several CRC32 operations over secret key K_W xored with sector number. Final whitening value is then applied to the whole plaintext sector with plain xor function. Unfortunately, both operations are not secure (IV is still partially predictable because for consecutive sectors we know which bits will change, and the whitening is just linear transformation, CRC32 is not cryptographically secure). The figure shows simplified attack with P (plaintext block) and C (ciphertext block). For more info see [TC4].

CBC mode – hidden disk watermark

- for sector S_n divisible by 4: S_n **xor** $S_{n+1} = S_{n+2}$ **xor** S_{n+3}
- whitening **W:** $W(K_w, A \text{ xor } B) = W(K_w, A) \text{ xor } W(K_w, B)$
- let's change the first plaintext blocks of sectors n .. n+3
  P_n and $P_{n+2} = 0$
  P_{n+1} and $P_{n+3} = S_n$ **xor** S_{n+1}

$C_n = E(K_E, P_n \text{ xor } IV_n) \text{ xor } W_n = \mathbf{E(K, K_{iv} \text{ xor } S_n)} \text{ xor } W_n$

$C_{n+1} = E(K_E, P_{n+1} \text{ xor } IV_{n+1}) \text{ xor } W_{n+1} = .. = \mathbf{E(K, K_{iv} \text{ xor } S_n)} \text{ xor } W_{n+1}$

- this allows to eliminate encryption and we can show that

$C_n \text{ xor } C_{n+1} = C_{n+2} \text{ xor } C_{n+3}$ **Watermark detection only from ciphertext!**

This attack allows construct a special plaintext, which if aligned to proper sectors can propagate special pattern (watermark) into ciphertext. Because filesystem aligns start of files to sector offset, we can just use specially formatted file in hidden disk and then search for the watermark on ciphertext device. As the exercise will show, this special file can be text file, so forcing user to store such file in the hidden disk is not too complicated (imagine mail or picture in browser cache).

Your Task

For the task use TrueCrypt installed in Virtual Machine (VeraCrypt no longer supports CBC mode). The exercise files are already copied to /home/task3 inside VM.

1. Use the provided TrueCrypt container in **/home/task3/tc_hidden_cbc.tc** and try to open it in TrueCrypt.
   The password to outer (decoy) volume is "**password**", password to hidden volume is "**hidden**".
2. Using the provided program **/home/task3/create_file <name>** (which implements attack above) generate special file and copy it to hidden TrueCrypt disk.
3. Dismount the TrueCrypt device and run **/home/task3/detect_file** over the TrueCrypt container image. It should find the pattern in special file and print the offset of this pattern.
4. See the create_file and detect_file source code (/home/task3/src) to better understand internal operation.

References

[TC4] sci-crypt, *TrueCrypt 4.0 Out [see example source code, message seems to be deleted...]* https://groups.google.com/forum/#!topic/sci.crypt/3DxOChZ0lrQ[1-25-false]