MIECT: Security 2019-20

Practical Exercise: Encrypted Storage

December 5, 2019 Due date: no date

Changelog

- v1.2 - Added references to commands that need to be executed by root.
- v1.1 - Fixed reference to file to look for patterns with the `strings` command.
- v1.0 - Initial version.

1 Introduction

Computers are frequently secure against on-line attacks, which target services and file permissions. This is enforced by the operating system kernel through the use of authentication mechanisms and file permissions, and effectively is able to keep an attacker away from accessing confidential data. However, this only provides protection against remote attackers. If a computer, such as a laptop, is stolen, or simply, the attacker has physical access to it, these security measures are insufficient. By booting from an external medium, such as a flash drive, or a CD, and by directly accessing the file system and block devices, all data will be available for inspection, or even tampering.

There are several ways of securing a computer from off-line attacks, which the BIOS password is an example. Without the proper secret, the computer can be restricted to boot, preventing external boot attacks. However, the system is still vulnerable to off-line attacks which target the storage media directly. That is, an hard disk can be removed from a computer, and its content analyzed. Moreover, if using swap memory space, some memory content, potentially containing private data, can be extracted.

Preventing this type of attacks requires the usage of cryptographic mechanisms in order to secure some of the stored data. Although not handled by this laboratory guide, due to the lack of proper hardware, a solution that can be considered is hardware-based encryption of the entire hard disk, handled transparently by the hard disk controller. One example of such product is the Seagate line of Self-Encrypting Drives (SED)[1].

The following sections will describe several methods to achieve (in a Linux system) security against off-line attacks, when using software solutions, ordered by level of scope.

2 File System Level

With the lowest scope is the technique which allows storing secure contents, by encrypting the content of sensitive files. With these solutions, the file system structure is available to an attacker, as well as some details of the stored files (permissions, size). However, it will not be easy to access the content, due to the use of an encryption algorithm.

[1] `https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp1299.pdf`

One of the first solutions was the Cryptographic File System (CFS) for Unix, BSD and Linux (circa 1993). Microsoft NTFS also supported encrypted files since the year 2000, as this feature, named Encrypting File System (EFS) was introduced in Windows 2000. Nowadays there are many solutions, being EncFS one of the most well known.

EncFS operates by providing an overlay over an existing directory, which can a remote mount point, so that all files written to that overlay are ciphered in real time. A great advantage is that when using EncFS, one unprivileged user can securely store files in shared, remote locations such as a SAMBA or NFS export. The drawback is that metadata is not ciphered. Therefore, file size, permissions, and file modification time will be publicly available.

The first step in using EncFS is to select two directories, one to store the encrypted files, and another to mount the encrypted file system. In this guide, please use `/opt/encfs-store` as the directory to store files, and `/opt/encfs-mount` as the directory to access the files:

```
mkdir -p /opt/encfs-store
mkdir -p /opt/encfs-mount
```

The next step is to activate the EncFS, by issuing[2] the next command (as root):

```
encfs /opt/encfs-store /opt/encfs-mount
```

During the setup process, choose `x` for greater insight regarding the methods used. After the process is complete, the new file system is available at `/opt/encfs-mount`. Create some files and directories in this mount point. Then observe the content of the directory `/opt/encfs-store`. You can unmount the EncFS by issuing `umount /opt/encfs-mount`.

Unmount the system, delete all files in `/opt/encfs-store` and repeat the process. This time, select a different set of options. In particular those related to the name of the files. Mount the file system and observe the result of creating files in the `/opt/encfs-store` directory.

3 Virtual Block Device

Increasing the level of security will require that file metadata is not available to others. One method of achieving this is by creating a file, which can be stored in a public location, and use that file as a secure block device. Block devices are devices such as flash drives and hard disks, which export a block-based interface. On top of these devices it is possible to create partitions, and file systems, such as the EXT4 file system. Although the EXT4 file system doesn't support online encryption, if the underlying block device transparently ciphers information, secure data storage can be achieved.

The purpose of a virtual block device, or loopback device, is to map a standard file as a block device. An interesting advantage of this method is that the software mapping the file can cipher information as it is written to the loopback device.

In order to create a loopback device, first it is required to create a file to be mapped. This file can be considered to be similar to a VDI or QCOW image used by VirtualBox.

Create a 100 MB file located at `/opt/secure-file`:

```
dd if=/dev/zero of=/opt/secure-file bs=1M count=100
```

Afterwards, map the file as a block device, and set that all operations should be ciphered/deciphered (execute these commands as root):

```
cryptsetup luksFormat /opt/secure-file
cryptsetup luksOpen /opt/secure-file secret-data
```

[2] you may need to install the `encfs` package

After this step, the node `/dev/mapper/secret-data` can be handled as a block device, and all accesses are directed to the `/opt/secure-file`. As with all block devices, a file system much be created, and then mounted. In this case, you should also create a directory named `/opt/secure-file-mount`:

```
mkdir -p /opt/secure-file-mount
mkfs.ext3 /dev/mapper/secret-data
mount /dev/mapper/secret-data /opt/secure-file-mount
```

If the process is complete, the file system can be accessed through the `/opt/secure-file-mount` directory. Copy some text files to this directory. As you can see, the file size is limited, and cannot be modified. This is a limitation of the method used. You can use the `strings` command over the file `/opt/secure-file` to check that the content is actually ciphered and pseudo-random.

You can unmount the file system by issuing:

```
umount /opt/secure-file-mount
```

And you can destroy the mapping attributed to the `/dev/mapper/secret-data` node by issuing (as root):

```
cryptsetup luksClose secret-data
```

After this step, the file is free from any binding and can be transported to another system, keeping all information secure.

4 Full Disk Encryption

The previous methods allowed to store private data in common folders. In the first case it is even possible to share the same storage between several non-colliding users. The next level of privacy is the use of full disk encryption so that all contents are kept private. While this method provides superior security, it is the less flexible[3] as encryption will be made at the level of an entire block storage device, or an entire partition. Access to data will be restricted until the proper key is provided, and all contents in the hard disk will be encrypted.

- The first step is to add a second disk to the Virtual Machine. This second disk will be ciphered.
- Then, the `cryptsetup` package must be installed with `apt-get`.
- Use `gparted` and create one partition.
- Format the partition to the LUKS format so that it can be used as a encrypted partition[4]:

  ```
  cryptsetup -v --cipher aes-xts-plain --key-size 256 \
      --hash sha1  --iter-time 1000 --use-urandom \
      --verify-passphrase luksFormat /dev/sdb1
  ```

- Create a directory named `/mnt/data`.
- Create a mapping to the encrypted partition, then a file system over it, and mount the file system to the `/mnt/data` directory:

  ```
  cryptsetup luksOpen  /dev/sdb1 data_crypt
  mkfs.ext4 /dev/mapper/data_crypt
  tune2fs -L "data_crypt" /dev/mapper/data_crypt
  mount /dev/mapper/data_crypt /mnt/data
  ```

- Edit `/etc/fstab` and add:

  ```
  /dev/mapper/data_crypt /mnt/data ext4 rw,noatime 0 1
  ```

[3] At least of the ones that will be used in this guide

[4] Be careful about the device that you format. In this case `/dev/sdb1` will be used, but your device may be a different one

After this step, the partition will be available at `/mnt/data` and all information written to it will be ciphered at a block level. Therefore, independently of the file system on top of it.

This process could also be applied to the root partition. However, the process has some complexity and is out of scope for this guide.

To close the device, the file system must be unmounted, and then the partition mapping removed:

```
umount /mnt/data
cryptsetup luksClose /dev/mapper/data_crypt
```

5 References

- https://www.arg0.net/encfs
- https://www.tldp.org/HOWTO/Cryptoloop-HOWTO
- https://gitlab.com/cryptsetup/cryptsetup