imagemounter Documentation

Release 1.5.1

Ralph Broenink, Peter Wagenaar

June 17, 2015

Contents

imagemounter is a command-line utility and Python package to ease the mounting and unmounting of EnCase, Affuse and dd disk images. It supports mounting disk images using xmount (with optional RW cache), affuse and ewfmount; detecting DOS, BSD, Sun, Mac and GPT volume systems; mounting Ext, UFS, LUKS and NTFS volumes; detecting (nested) LVM volume systems and mounting its subvolumes; and reconstructing RAID arrays.

In its default mode, imagemounter will try to start mounting the base image on a temporary mount point, detect the volume system and then mount each volume seperately. If it fails finding a volume system, it will try to mount the entire image as a whole if it succeeds in detecting what it actually is.

CHAPTER 1

Important notes

Not all combinations of file and volume systems have been tested. If you encounter an issue, please try to change some of your arguments first, before creating a new GitHub issue.

Please note that many Linux based operating systems will try to mount LVMs for you. Although imagemounter tries to circumvent this automation, if you are unable to properly unmount, you should try to unmount through the interface of your OS first. Another useful command is *vgchange -a n* to disable all LVMs currently active (only use if you are not using a LVM for your own OS!).

With *imount –clear* you can clear MOST temporary files and mounts, though this will not clean everything. If you used *–pretty* this tool can't do anything for you. It is therefore recommended to first try and mount your image without *–pretty*, to allow you to easily clean up if something crashes.

CHAPTER 2

Contents

2.1 Installation

If you need an installation with full support, including all optional dependencies, you could use the following commands:

```
apt-get install python-setuptools xmount ewf-tools afflib-tools sleuthkit lvm2 mdadm cryptsetup
pip install imagemounter
```

2.1.1 Python packages

This package does not require other packages, though the `termcolor` package is recommended if you are using the **`imount`** command line utility with the `--color` argument.

If you wish to use `pytsk3` support, you require *python-dev* and *libtsk-dev*. For compilation, the *build-essential* package from your distribution is also required. After that, you can easily install the `pytsk3` package from PyPI (**`pip`** requires the `--pre` flag to allow installing the package).

2.1.2 Other dependencies

This package highly depends on other utilities to be present on your system. For a full installation, you require the following tools:

- Mount tools
  - **`xmount`**
  - **`ewfmount`**, part of *ewf-tools* package, see note below
  - **`affuse`**, part of *afflib-tools* package
  - **`vmware-mount`**, part of VMware Workstation
- Volume detection
  - **`mmls`**, part of *sleuthkit* package
  - `pytsk3`
- Statistics, e.g. last mountpoint of volumes
  - **`fsstat`**, part of *sleuthkit* package
- LVM volumes

  - **lvm** et al, all part of *lvm2* package
- RAID arrays
  - **mdadm**
- LUKS volumes
  - **cryptsetup**

A basic installation contains at least one of the mount tools. Highly recommended is also *fsstat*, others are required for specific file system types.

2.1.3 ewfmount on Ubuntu 13.10

Due to a bug with *ewf-tools* in Ubuntu <=13.10, it may be that **ewfmount** is not properly provided. This bug has been resolved in Ubuntu 14.04. If you are using Ubuntu 13.10, you can install *ewf-tools* with **ewfmount** as follows:

1. Download a recent build of *ewf-tools* from https://launchpad.net/ubuntu/+source/libewf/20130416-2ubuntu1 (choose your arch under 'Builds' and download all deb files under 'Built files')
2. Execute `sudo apt-get install libbfio1`
3. Execute `sudo dpkg -i ewf-tools_* libewf2_*`

2.2 Command-line usage

One of the core functionalities of `imagemounter` is the command-line utility **imount** that eases the mounting and unmounting of different types of disks and volumes. In its most basic form, the utility accepts a positional argument pointing to a disk image, disk or volume, e.g.:

```
imount disk.E01
```

Multiple files can be passed to this command, allowing the mounting of volume systems that span multiple disks, which can be useful for those wishing to reconstruct a system that entailed multiple disks or for reconstructing RAID arrays.

By default, **imount** will mount each single volume in `/tmp` and wait until you confirm an unmount operation. Common usage is therefore to keep **imount** running in a separate window and perform other operations in a second window.

2.2.1 Arguments

The **imount** utility requires one (or more) positional arguments and offers the ability to pass several optional arguments.

<image> [<image> ...]

The positional argument(s) should provide the path(s) to the disk images you want to mount. Many different formats are supported, including the EnCase evidence format, split dd files, mounted hard drives, etc. In the case of split files, you can refer to the folder containing these files.

If you specify more than one file, all files are considered to be part of the same originating system, which is relevant for the `--reconstruct` command-line option.

Arguments that immediately exit

Some useful facilities.

--help
-h

Shows a help message and exits.

--version

Shows the current version and exits.

--clean

Option that will try to identify leftover files from previous **imount** executions and try to delete these. This will, for instance, clean leftover `/tmp/im_...` mounts and mountpoints. This command will allow you to review the actions that will be taken before they are done.

CLI behaviour

The next four command-line options alter the behaviour of the **imount** utility, but does not affect the behaviour of the underlying *imagemounter* module.

--color
-c

Colorizes the output. Verbose message will be colored blue, for instance. Requires the `termcolor` package.

--wait
-w

Pauses the execution of the program on all warnings.

--keep
-k

Skips the unmounting at the end of the program.

--verbose
-v

Show verbose output

Additional features

This command-line option enables an additional and useful feature.

--reconstruct
-r

Attempts to reconstruct the full filesystem tree by identifying the last mountpoint of each identified volume and bindmounting this in the previous root directory. For instance, if volumes have previously been mounted at `/` , `/var` and `/home` ; `/var` and `/home` will be bind-mounted in `/` , providing you with a single filesystem tree in the mount location of `/` that is easily traversible.

This only works with Linux-based filesystems and only if `/` can be identified.

Implies *--stats*.

Mount behaviour

These arguments alter some pieces of the mount behaviour of *imagemounter*, mostly to ease your work.

--mountdir `<directory>`

-md <directory>
: Specifies the directory to place volume mounts. Defaults to a temporary directory.

--pretty
-p
: Uses pretty names for volume mount points. This is useful in combination with `--mountdir`, but you should be careful using this option. It does not provide a fallback when the mount point is not available or other issues arise. It can also not be cleaned with `--clean`.

--read-write
-rw
: Will use read-write mounts. Written data will be stored using a local write cache.

  Implies `--method xmount`.

Advanced options

While `imagemounter` will try to automatically detect as much as possible, there are some cases where you may wish to override the automatically detected options. You can specify which detection methods should be used and override the volume system and file system types if needed.

--method <method>
-m <method>
: Specifies the method to use to mount the base image(s). Defaults to automatic detection, though different methods deliver different results. Available options are *xmount*, *affuse* and *ewfmount* (defaulting to *auto*).

  If you provide *dummy*, the base is not mounted but used directly.

--detection <method>
-d <method>
: Specifies the volume detection method. Available options are *pytsk3*, *mmls* and *auto*, which is the default. Though *pytsk3* and *mmls* should in principle deliver identical results, *pytsk3* can be considered more reliable as this uses the C API of The Sleuth Kit (TSK). However, it also requires `pytsk3` to be installed, which is not possible with Py3K.

--vstype <type>
: Specifies the type of the volume system, defaulting to *detect*. However, detection may not always succeed and valid options are *dos*, *bsd*, *sun*, *mac*, *gpt* and *dbfiller*, though the exact available options depend on the detection method and installed modules on the operating system.

--fsfallback <type>
: Specifies a fallback option for the filesystem of a volume if automatic detection fails. Available options are *ext*, *ufs*, *ntfs*, *luks*, *lvm* and *unknown*, with the latter simply mounting the volume without specifying type.

--fsforce
: Forces the use of the filesystem type specified with `--fsfallback` for all volumes. In other words, disables the automatic filesystem detection.

--fstypes <types>
: Allows the specification of filesystem type for each volume separately. You can use subvolumes, examples including:

```
1=ntfs
2=luks,2.0=lvm,2.0.1=ext
```

Advanced toggles

imount has some facilities that automatically detect some types of disks and volumes. However, these facilities may sometimes fail and can be disabled if needed.

--stats
--no-stats

With stats rerieval is enabled, additional volume information is obtained from the **fsstat** command. This could possibly slow down mounting and may cause random issues such as partitions being unreadable. However, this additional information will probably include some useful information related to the volume system and is required for commands such as `--reconstruct`.

Stats retrieval is enabled by default, but `--stats` can be used to override `--no-stats`.

--raid
--no-raid

By default, a detection is ran to detect whether the volume is part of a (former) RAID array. You can disable the RAID check with `--no-raid`. If you provide both `--raid` and `--no-raid`, `raid` wins.

--single
--no-single

imount will, by default, try to detect whether the disk that is being mounted, contains an entire volume system, or only a single volume. If you know your volumes are not single volumes, or you know they are, use `--no-single` and `--single` respectively.

Where `--single` forces the mounting of the disk as a single volume, `--no-single` will prevent the identification of the disk as a single volume if no volume system is found.

2.3 Python interface

While **imount** heavily utilizes the Python API of `imagemounter`, this API is also available for other classes.

2.3.1 Data structure

The basic structure of `imagemounter` is the `imagemounter.ImageParser` class, which provides access to underlying `imagemounter.Disk` and `imagemounter.Volume` objects. Each file name passed to a new `imagemounter.ImageParser` object results in one `imagemounter.Disk` object. `imagemounter.Volume` objects are created by analysis of the `Disk` object (each volume generates one object, even if it is not mountable), and each `imagemounter.Volume` can have one or more subvolumes.

For instance, a LUKS volume may contain a LVM system that contains a Ext volume. This would create a `Disk` with a `Volume` containing a `Volume` which contains the actual Ext `Volume`.

Most operations are managed on a `Volume` level, although RAIDs (and volume detection) are managed on a `Disk` level and reconstruction is performed on a `ImageParser` level. This means the following main parts make up the Python package:

- `imagemounter.ImageParser`, maintaining a list of Disks, providing several methods that are carried out on all disks (e.g. mount) and reconstruct.
- `imagemounter.Disk`, which represents a single disk iamge and can be mounted, added to RAID, and detect and maintain volumes. It is also responsible for maintaining the write cache.
- `imagemounter.Volume`, which can detect its own type and fill its stats, can be mounted, and detect LVM (sub)volumes.

All three classes maintain an `init()` method that yields the volumes below it. You should call `clean()` on the parser as soon as you are done; you may also call `unmount()` on separate volumes or disks, which will also unmount all volumes below it. Warning: unmounting one of the RAID volumes in a RAID array, causes the entire array to be unmounted.

2.3.2 Reference

If you utilize the API, you typically only require the `ImageParser` object, e.g.:

```
parser = ImageParser(['/path/to/disk'])
for v in parser.init():
    print v.size
root = parser.reconstruct()
print root.mountpoint
parser.clean()
```

The best example of the use of the Python interface is the **`imount`** command. The entirety of all methods and attributes is documented below.

class `imagemounter.`**`ImageParser`**(*paths*, *out=None*, *verbose=False*, *color=False*, ***args*)

Root object of the `imagemounter` Python interface. This class should be sufficient allowing access to the underlying functions of this module.

Instantiation of this class does not automatically mount, detect or analyse `Disk` s, though it initialises each provided path as a new `Disk` object.

Parameters

- **`paths`** (*iterable*) – list of paths to base images that should be mounted
- **`out`** – location where verbose output should be written, defaulting to `sys.stdout`
- **`verbose`** (*bool*) – indicates whether verbose output should be written
- **`color`** (*bool*) – indicates whether verbose output should be colored
- **`args`** – arguments that should be passed down to `Disk` and `Volume` objects

`init`(*single=None*, *raid=True*)

Handles all important disk-mounting tasks, i.e. calls the `Disk.init()` function on all underlying disks. It yields every volume that is encountered, including volumes that have not been mounted.

Parameters

- **`single`** (*bool|None*) – indicates whether the `Disk` should be mounted as a single disk, not as a single disk or whether it should try both (defaults to `None`)
- **`raid`** (*bool*) – indicates whether RAID detection is enabled

Return type generator

`reconstruct`()

Reconstructs the filesystem of all volumes mounted by the parser by inspecting the last mount point and bind mounting everything.

Returns None on failure, or the root `Volume` on success

`clean`(*remove_rw=False*)

Cleans all volumes of all disks (`Volume.unmount()`) and all disks (`Disk.unmount()`). Volume errors are ignored, but returns immediately on disk unmount error.

Parameters **`remove_rw`** (*bool*) – indicates whether a read-write cache should be removed

Returns whether the command completed successfully

Return type boolean

static **`force_clean`**(*execute=True*)

Executes a full clean-up of any left-over traces of previous runs of *`imagemounter`*. This detection is separate from any program execution and may therefore detect not everything or detect too much.

Parameters **`execute`** (*bool*) – indicates whether the actions should be executed or only returned

Returns list of all commands (to be) executed

Most methods above, especially *`init()`*, handle most complicated tasks. However, you may need some more fine-grained control over the mount process, which may require you to use the following methods. Each of these methods passes their activities down to all disks in the parser and return whether it succeeded.

`rw_active`()

Indicates whether a read-write cache is active in any of the disks.

Return type bool

`get_volumes`()

Gets a list of all volumes of all disks, concatenating *`Disk.get_volumes()`* of all disks.

Return type list

`mount_disks`()

Mounts all disks in the parser, i.e. calling *`Disk.mount()`* on all underlying disks. You probably want to use *`init()`* instead.

Returns whether all mounts have succeeded

Return type bool

`mount_raid`()

Creates a RAID device and adds all devices to the RAID array, i.e. calling *`Disk.add_to_raid()`* on all underlying disks. Should be called before *`mount_disks()`*.

Returns whether all disks were successfully added

Return type bool

`mount_single_volume`()

Detects the full disk as a single volume and yields the volume. This calls *`Disk.mount_single_volume()`* on all disks and should be called after *`mount_disks()`*

Return type generator

`mount_multiple_volumes`()

Detects volumes in all disks (all mounted as a volume system) and yields the volumes. This calls *`Disk.mount_multiple_volumes()`* on all disks and should be called after *`mount_disks()`*.

Return type generator

`mount_volumes`(*single=None*)

Detects volumes (as volume system or as single volume) in all disks and yields the volumes. This calls *`Disk.mount_multiple_volumes()`* on all disks and should be called after *`mount_disks()`*.

Return type generator

For completeness, this is a list of all attributes of *`ImageParser`*:

`disks`

List of all *`Disk`* objects.

paths
out
verbose
verbose_color
args

See the constructor of `ImageParser`.

class `imagemounter.`**`Disk`**(*parser*, *path*, *offset=0*, *vstype=u'detect'*, *read_write=False*, *method=u'auto'*, *detection=u'auto'*, *multifile=True*, *index=None*, ***args*)

Representation of a disk, image file or anything else that can be considered a disk.

Instantiation of this class does not automatically mount, detect or analyse the disk. You will need the `init()` method for this.

Parameters

- **parser** (`ImageParser`) – the parent parser
- **offset** (*int*) – offset of the disk where the volume (system) resides
- **vstype** (*str*) – the volume system type
- **read_write** (*bool*) – indicates whether the disk should be mounted with a read-write cache enabled
- **method** (*str*) – the method to mount the base image with
- **detection** (*str*) – the method to detect volumes in the volume system with
- **multifile** (*bool*) – indicates whether `mount()` should attempt to call the underlying mount method with all files of a split file when passing a single file does not work
- **args** – arguments that should be passed down to `Volume` objects

init(*single=None*, *raid=True*)

Calls several methods required to perform a full initialisation: `mount()`, `add_to_raid()` and `mount_volumes()` and yields all detected volumes.

Parameters

- **single** (*bool|None*) – indicates whether the disk should be mounted as a single disk, not as a single disk or whether it should try both (defaults to `None`)
- **raid** (*bool*) – indicates whether RAID detection is enabled

Return type generator

unmount(*remove_rw=False*)

Removes all ties of this disk to the filesystem, so the image can be unmounted successfully. Warning: this method will destruct the entire RAID array in which this disk takes part.

The following methods are only required if you want some fine-grained control, typically if you are not using `init()`.

rw_active()

Indicates whether anything has been written to a read-write cache.

get_fs_path()

Returns the path to the filesystem. Most of the times this is the image file, but may instead also return the MD device or loopback device the filesystem is mounted to.

Return type str

get_raw_path()

Returns the raw path to the mounted disk image, i.e. the raw `.dd`, `.raw` or `ewf1` file.

Return type str

get_volumes()
Gets a list of all volumes in this disk, including volumes that are contained in other volumes.

mount()
Mounts the base image on a temporary location using the mount method stored in *method*. If mounting was successful, *mountpoint* is set to the temporary mountpoint.

If *read_write* is enabled, a temporary read-write cache is also created and stored in *rwpath*.

Returns whether the mounting was successful

Return type bool

mount_volumes(*single=None*)
Generator that detects and mounts all volumes in the disk.

If *single* is `True`, this method will call `mount_single_volumes()`. If *single* is False, only *mount_multiple_volumes()* is called. If *single* is None, *mount_multiple_volumes()* is always called, being followed by *mount_single_volume()* if no volumes were detected.

mount_multiple_volumes()
Generator that will detect volumes in the disk file, generate *Volume* objects based on this information and call *init()* on these.

mount_single_volume()
Mounts a volume assuming that the mounted image does not contain a full disk image, but only a single volume.

A new *Volume* object is created based on the disk file and *init()* is called on this object.

This function will typically yield one volume, although if the volume contains other volumes, multiple volumes may be returned.

is_raid()
Tests whether this image (was) part of a RAID array. Requires **mdadm** to be installed.

add_to_raid()
Adds the disk to a central RAID volume.

This function will first test whether it is actually a RAID volume by using *is_raid()* and, if so, will add the disk to the array via a loopback device.

Returns whether the addition succeeded

The following attributes are also available:

name
Pretty name of the disk.

index
Disk index. May be None if it is the only disk of this type.

mountpoint
The mountpoint of the disk, after a call to *mount()*.

rwpath
The path to the read-write cache, filled after a call to *mount()*.

volumes
List of all direct child volumes of this disk, excluding all subvolumes. See *get_volumes()*.

volume_source
The source of the volumes of this disk, either *single* or *multi*, filled after a call to *mount_volumes()*.

loopback
md_device
Used for RAID support.

parser
path
offset
vstype
read_write
method
detection
multifile
args
See the constructor of `Disk`.

class `imagemounter.`**`Volume`**(*disk=None*, *stats=False*, *fsforce=False*, *fsfallback=None*, *fstypes=None*, *pretty=False*, *mountdir=None*, ***args*)

Information about a volume. Note that every detected volume gets their own Volume object, though it may or may not be mounted. This can be seen through the `mountpoint` attribute – if it is not set, perhaps the `exception` attribute is set with an exception.

Creates a Volume object that is not mounted yet.

Parameters

- **disk** (`Disk`) – the parent disk
- **stats** (*bool*) – indicates whether `init()` should try to fill statistics
- **fsforce** (*bool*) – indicates whether the file system type in *fsfallback* should be used for all file systems
- **fsfallback** (*str*) – the file system type to use when automatic detection fails
- **fstypes** (*dict*) – dict mapping volume indices to file system types to (forcibly) use
- **pretty** (*bool*) – indicates whether pretty names should be used for the mountpoints
- **mountdir** (*str*) – location where mountpoints are created, defaulting to a temporary location
- **args** – additional arguments

init(*no_stats=False*)
Generator that mounts this volume and either yields itself or recursively generates its subvolumes.

More specifically, this function will call `fill_stats()` (iff *no_stats* is False), followed by `mount()`, followed by a call to `detect_mountpoint()`, after which `self` is yielded, or the result of the `init()` call on each subvolume is yielded

unmount()
Unounts the volume from the filesystem.

The following methods offer some more information about the volume:

get_description(*with_size=True*)
Obtains a generic description of the volume, containing the file system type, index, label and NTFS version. If *with_size* is provided, the volume size is also included.

get_safe_label()
Returns a label that is safe to add to a path in the mountpoint for this volume.

get_size_gib()
Obtains the size of the volume in a human-readable format (i.e. in TiBs, GiBs or MiBs).

get_volumes()
: Recursively gets a list of all subvolumes and the current volume.

These functions offer access to some internals:

get_fs_type()
: Determines the FS type for this partition. This function is used internally to determine which mount system to use, based on the file system description. Return values include *ext*, *bsd*, *ntfs*, *lvm* and *luks*.

get_raw_base_path()
: Retrieves the base mount path of the volume. Typically equals to `Disk.get_fs_path()` but may also be the path to a logical volume. This is used to determine the source path for a mount call.

mount()
: Based on the file system type as determined by `get_fs_type()`, the proper mount command is executed for this volume. The volume is mounted in a temporary path (or a pretty path if `pretty` is enabled) in the mountpoint as specified by `mountpoint`.

  If the file system type is a LUKS container, `open_luks_container()` is called only. If it is a LVM volume, `find_lvm_volumes()` is called after the LVM has been mounted. Both methods will add subvolumes to `volumes`

  Returns boolean indicating whether the mount succeeded

bindmount(*mountpoint*)
: Bind mounts the volume to another mountpoint. Only works if the volume is already mounted. Note that only the last bindmountpoint is remembered and cleaned.

  Returns bool indicating whether the bindmount succeeded

fill_stats()
: Using **fsstat**, adds some additional information of the volume to the Volume.

detect_mountpoint()
: Attempts to detect the previous mountpoint if this was not done through `fill_stats()`. This detection does some heuristic method on the mounted volume.

find_lvm_volumes(*force=False*)
: Performs post-mount actions on a LVM. Scans for active volume groups from the loopback device, activates it and fills `volumes` with the logical volumes.

  If *force* is true, the LVM detection is ran even when the LVM is not mounted on a loopback device.

open_luks_container()
: Command that is an alternative to the `mount()` command that opens a LUKS container. The opened volume is added to the subvolume set of this volume. Requires the user to enter the key manually.

  Returns the Volume contained in the LUKS container, or None on failure.

The following details may also be available as attributes:

size
: The size of the volume in bytes.

offset
: The offset of the volume in the disk in bytes.

index
: The index of the volume in the disk. If there are subvolumes, the index is separated by periods, though the exact format depends on the detection method and its format.

flag
: Indicates whether this volume is allocated (*alloc*), unallocated (*unalloc*) or a meta volume (*meta*).

fsdescription
A description of the file system type.

lastmountpoint
The last mountpoint of this volume. Set by *fill_stats()* or *detect_mountpoint()* and only available for UFS and Ext volumes.

label
The volume label as detected by *fill_stats()*.

version
The volume version as detected by *fill_stats()*.

fstype
The volume file system type as detected by *fill_stats()*.

mountpoint
The mountpoint of the volume after *mount()* has been called.

bindmountpoint
The mountpoint of the volume after *bindmount()* has been called.

loopback
The loopback device used by the volume after *mount()* (or related methods) has been called.

exception
Contains an exception that occurred during a call to *mount()*.

was_mounted
Boolean indicating that the volume has successfully been mounted during its lifetime.

volumes
parent
volumes contains a list of all subvolumes of this volume; *parent* contains the parent volume (if any).

volume_group
lv_path
Attributes used for LVM support

luks_path
Attribute used for LUKS support

disk
stats
fsforce
fsfallback
fstypes
pretty
mountdir
args
See the constructor of *Volume*.

Index