AFF4 imager Documentation

Release 1.0

Michael Cohen

Feb 08, 2018

Contents

AFF4 is an advanced, open forensic imaging format. The format has been standardized in the AFF4StandardSpecification.

CHAPTER 1

Why use AFF4?

The AFF4 format is specifically designed to support a wide range of use cases in forensic evidence capture and preservation. Some of the benefits of using AFF4 over another proprietary format are

- The AFF4 format is an opened standard with many open source implementations in a variery of languages. The format itself is very simple and so it will always be possible to extract data even without an proper implementation.
- AFF4 storage is designed around the widely used Zip file format. This means that you can use typical Zip recovery utilities to repair corrupted images, inspect the content of images etc.
- AFF4 supports multiple streams within the same volume. This allows multiple sources of evidence to be captured into the same case file. For example, related memory, disk and logical file streams can be handled simultaneously.
- AFF4 supports sparse streams (using the Map stream type). This is useful for representing memory images (which might have gaps in them) as well as simply representing read errors (rather than simply zero padding them).
- AFF4 supports arbitrary metadata as RDF statements.

CHAPTER 2

The AFF4 C/C++ Library

This project is a C++ implementation of the AFF4 standard. The library allows reading, writing and verifying AFF4 images, and can be easily integrated into any C++ project.

See the documentation for developers.

CHAPTER 3

The AFF4 Imager

The main tool implemented in this project is the *aff4imager* tool. The tool provides a basic framework for implementing imaging tools, and therefore it is also the basis for a number of other tools, such as linpmem, winpmem and osxpmem (A memory acquisition suite). All the below commands should also work on these tools as well.

You can download the latest release of the aff4 imager through the project's page. Releases are also statically built for their respective platforms in order to ensure that they can be easily deployed with minimal system requirements - even on very old systems.

The following section describes how to use the imager effectively. You can get some helpful messages from the images itself when run with the *–help* flag.

3.1 Acquiring files.

The AFF4 imager can acquire multiple files into a new AFF4 volume. These can be devices (such as disks using /dev/sda) or logical files.

1. Acquiring a disk image:

```
affimager -i /dev/sda -o /tmp/output.aff4
```

   Note that by default the aff4 imager will append new streams to the output volume if it already exists. This is useful for appending more relevant evidence after the initial acquisition is completed. If you dont want this behaviour you can specify the -t (–truncate) flag to truncate the output volume before adding the new stream.

2. Acquiring multiple logical files:

```
affimager -i /bin/* -o /tmp/output.aff4
```

   Using glob expressions as input files will be expanded as required to include all filenames matching the globs. This works also on Windows which does not expand globs on the shell.

3. Acquiring files newer than 30 days:

```
find /usr/bin/ -ctime -30 | aff4imager -i @ -o /tmp/output.aff4
```

Using a single @ as the input filename, makes the aff4 imager read the list of files to acqiure from stdin. This allows for more sophisticated pre-processing and makes it easier to acquire files with spaces or special characters in their names (without having to worry about shell escapes). In the above example we use the find unix command to identify files newer than 30 days and also add them to the image.

4. Enabling multiple threads:

```
affimager -i /dev/sda -o /tmp/output.aff4 --threads 6
```

The aff4 imager uses a single thread by default, but if your machine has more cores, then you will see vastly better performance by allowing more threads to run. This is particularly important when using the default compression of the zlib compressor which needs more CPU resources.

5. Enabling snappy compression:

```
affimager -i /dev/sda -o /tmp/output.aff4 --compression snappy
```

The Snappy compression engine is much faster than the default zlib but trades off compression size. Enabling snappy compression will result in slightly larger images but should complete faster.

6. Splitting an image into multiple volumes:

```
affimager -i /dev/sda -o /tmp/output.aff4 --split 650m
```

Some images are very large. By enabling splitting images it is possible to restrict the maximum size of each volume. The imager will close off each volume as it is done with it, and so you can start uploading, archiving each volume as soon as it is finished. Note that the same stream may be split across one or more volumes so you will need all volumes to properly extract the stream.

7. Acquiring into standard output:

```
./aff4imager -i /bin/* -o - | gsutil cp - gs://rekall-test/test.aff4
```

If the output filename is specifies as a single dash ("-"), the imager writes the AFF4 volume to stdout. This allows the image to be piped to a different tool. The example above streams the image directly to a cloud storage bucket without needing to write a temporary local copy.

3.2 Inspecting AFF4 volumes.

AFF4 volumes are based around the common Zip compression standard (for large volumes we use Zip64 extensions). Therefore it is possible to examine AFF4 volumes using common zip utilities:

```
 unzip -l /tmp/test.aff4
Archive:  /tmp/test.aff4
aff4://c7c60030-cc3e-43a6-b5d1-1551b29c9918
Length      Date    Time    Name
---------  ---------- -----   ----
189432     2018-01-15 12:50   usr/bin/mksquashfs
951952     2018-01-15 12:50   usr/bin/x86_64-w64-mingw32-cpp
...

50929      2018-01-15 12:50   information.turtle
...
```

```
---------                       -------
315748394                       327 files
```

We can see that each volume has a unique URN, and it contains a file called "information.turtle". This file contains the AFF4 metadata for the volume as an RDF turtle file.

We can get the aff4 imager to display the metadata in the volume using the -V flag:

```
 aff4imager -V /tmp/test.aff4
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix aff4: <http://aff4.org/Schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<aff4://2d874721-267b-40fb-ac20-7bc22a8af883/proc/kallsyms>
  aff4:original_filename "/proc/kallsyms"^^xsd:string .

<aff4://2d874721-267b-40fb-ac20-7bc22a8af883/proc/kcore>
  aff4:category <http://aff4.org/Schema#memory/physical> ;
  aff4:stored <aff4://2d874721-267b-40fb-ac20-7bc22a8af883> ;
  a aff4:Image, aff4:Map .

<aff4://2d874721-267b-40fb-ac20-7bc22a8af883/proc/kcore/data>
  aff4:chunkSize 32768 ;
  aff4:chunksInSegment 1024 ;
  aff4:compressionMethod <https://www.ietf.org/rfc/rfc1950.txt> ;
  aff4:size 8531652608 ;
  aff4:stored <aff4://2d874721-267b-40fb-ac20-7bc22a8af883> ;
  a aff4:ImageStream .
```

Note that if we have multiple volumes (as in a split volume set) we should list all volumes as parameters to -V.

In the above output we see some interesting artifacts of the AFF4 format:

1. All streams within the AFF4 volume have a unique URN. The imager creates the URNs based on their original filename, but this is just a convenience. The imager also stores the original filename (which might contain backslashes on windows).
2. Smaller files (e.g. */proc/kallsyms*) are stored as AFF4 Segments which are just regular zip archive members. This means you can extract Smaller files using normal zip tools.
3. Larger files are stored as AFF4 ImageStream. This type of storage chunks the file data into 32kb chunks, and stores groups of chunks in their own zip segment.
4. Finally sparse images (such as memory images) are stores as an AFF4 Map. The map does not actually store any data itself (the data is stored by the stream */proc/kcore/data*) but it specifies a transformation of its underlying stream.

Finally using the -l flag enables a listing of all Image streams from the volume.

3.3 Extracting streams from AFF4 volumes.

To extract streams from an AFF4 volume we use the *-e* flag.

1. Extract streams by using wild cards:

```
aff4imager -e '*/kallsyms' --export_dir /tmp/export/ /tmp/test.aff4
```

By default export directory is the current directory. The imager will create a directory structure under the export directory which contains all the matched files.

2. Extract streams from stdin:

```
aff4imager -l /tmp/test.aff4 | grep kcore | \
/aff4imager -e @ --export_dir /tmp/export/ /tmp/test.aff4
```

3.3.1 For developers.