AFF and AFF4: Where We Are, Where We are Going, and Why it Matters to You

Simson L. Garfinkel
Associate Professor, Naval Postgraduate School

- 13:45
  June 9, 2010
  Sleuth Kit and Open Source Forensics Conference

NPS is the Navy's Research University.

Location: Monterey, CA   Campus Size: 627 acres

Students: 1500

- US Military (All 5 services)
- US Civilian (Scholarship for Service & SMART)
- Foreign Military (30 countries)

Schools:

- Business & Public Policy
- Engineering & Applied Sciences
- Operational & Information Sciences
- International Graduate Studies

We have programs for:

- US Government Employees
- Contractors
- Master's Students (Scholarship For Service)

"DEEP" — Current Research

AFF & Real Data Corpus

- http://afflib.org/
- http://digitalcorpora.org/

Automated metadata extraction and exploitation (XML & ARFF)

- fiwalk tool chain; redaction program;

Automated Ascription of Exploited Data

Sector Discrimination and Random Sampling

Goals of this talk

Present AFF history and Roadmap

- AFFLIB
- AFF4

API Layer — *interface* to analysis programs.

Schema Layer — *structure* of stored data

Bit-level layer — dictates *how* data is stored

Introduce Digital Forensics XML

<fileobject>

- fiwalk
- fiwalk.py

Promote Tools that are available to download NOW!

- frag_find
- bulk_extractor

NPS PRAESTANTIA PER SCIENTIAM 1909

AFFLIB v1-3

AFF was designed for large-scale *drive imaging* and *archiving*

In 1998 I started the "Drives Project."

- Looking for data on used computer equipment.

Between 1998-2005 I purchased 250 drives:

- Serial number info captured with **atacontrol**
- Drives imaged with **dd**
- Images stored in **raw** format, eventually compressed with **gzip**
- Good enough for my 2005 PhD Thesis.

In 2005 I started "Phase 2" of the project.

- Goal: Increase corpora size to 2500 drives.
- Development of new forensic techniques for LE & IC

Question: *How to store the disk images?*

NPS PRAESTANTIA PER SCIENTIAM

There were not many choices in 2005 for disk images.

EnCase Format

- Proprietary; no open source implementation. (libewf released in 2006)
- 2GB size limit created a management nightmare. (**FILE.E01, FILE.E02, FILE.E03…)**
- No provision for encryption or digital signatures.
  - *Encryption — needed for privacy, security, & IRB approval*
  - *Digital Signatures — to enable capture by "trusted hardware."*

Other proprietary formats:

- IXimager and ILook Investigator
- ProDiscover Image File Format
- SafeBack
- Vogon International's SDi32

PyFlag "Seekable gzip"

- Open source, but not implemented anywhere except PyFlag.
- No obvious way to store metadata

We decided to create AFF — the Advanced Forensic Format

Format Goals:

- Open Format — All bits clearly defined and documented.
- Excellent Compression
- One image file per physical disk
- Support Encryption
  - *Password-based private key*
  - *Certificate-based public key*

Implementation Goals

- Multi-platform: Windows, MacOS, Linux, FreeBSD, etc.
- Open Implementation — No licensing fees.
- Easy to instrument — enable research in computer forensics

NPS
PRAESTANTIA PER SCIENTIAM

AFF v1 has three distinct layers.

API Layer — *interface* to analysis programs.

Schema Layer — *structure* of stored data

Bit-level layer — dictates *how* data is stored

NPS PRAESTANTIA PER SCIENTIAM 1909

API Layer:
designed for easy integration into existing programs

API Layer — *interface* to analysis programs.

Simple interface:

```
AFFILE *af = af_open()

af_seek(af,pos,SEEK_SET);

af_read(af,buf,sizeof(buf));

af_close(af)
```

AFF stores all data as name/value pairs

The "schema" is standardized names for forensic data.

`sectorsize` — *Number of bytes per sector — 0x00000200 (512)*

`imagesize` — *Number of bytes in the logical image — 0x1000000000 (64GiB)*

`device_sn` — *Serial number of the device — "WCAM9J939319"*

`device_firmware` — *Drive capabilities*

Schema Layer — *structure* of stored data

Forensic data is stored in "pages"

- Page size is determined when image is created
- Default page: 16MiB
- Pages can be encrypted with: NULL, RAW, ZLIB, LZMA, etc.
- Each page has a name: "page0", "page1", "page2" …

The bit-level layer dictates how data is stored.

AFFLIB can store name/value pairs in different ways.

- AFF file

  - *Series of named segments, each with a HEAD; LENGTH; DATA; FOOT*
  - *Easy to recover in the event of corruption, off-track writes, etc.*
- AFD file
  - *Multiple AFF files in a single directory*
- AFF XML
- Amazon S3
- VMDK (via QEMU disk layer)

Bit-level layer — dictates *how* data is stored

AFFLIB also supports "legacy" formats:

- RAW, SPLIT RAW, EnCase E01 (libewf)

A simple example: creating a 64K blank disk image

```
#include <afflib/afflib.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc,char **argv)
{
    u_char buf[65536];
    memset(buf,0,sizeof(buf));

    AFFILE *af = af_open("file.aff",O_RDWR|O_CREAT,0777);
    af_write(af,buf,sizeof(buf));
    af_close(af);
    return(0);
}
g++ -odemo -I/usr/local/include demo.cpp -lafflib
```

Creates:

```
$ ls -l file.aff
-rwxr-xr-x  1 simsong  staff  820 May 31 08:20 file.aff*
$
```

The "afinfo" command shows the segemnts.

```
$ afinfo -a file.aff
file.aff is a AFF file
                                             data
Segment                          arg       length    data
=======                    =========     ========    ====
badflag                            0          512  BAD SECTOR.......U.8....}...Wj
badsectors                         2            8  = 0 (64-bit value)
afflib_version                     0            7  "3.5.8"
creator                            0            5  a.out
aff_file_type                      0            3  AFF
pagesize                    16777216            0
page0                             51            4  ....
imagesize                          2            8  = 65536 (64-bit value)

Total segments:                  8   (8 real)
  Page  segments:                1
  Hash  segments:                0
  Signature segments:            0
  Null segments:                 0
  Empty segments:                0

Total data bytes in segments: 547
Total space in file dedicated to segment names: 73
Total overhead for 8 segments: 192 bytes (8*(16+8))
Overhead for AFF file header: 8 bytes
$
```

AFFLIB v3 added encryption & digital signatures

Encryption: each segment can be encrypted with a 256-bit AES key.

- AFFLIB automatically encrypts & decrypts each segment on read if possible.

Key can be specified as:

- passphrase that decrypts an **`afkey_aes256`** segment.
- X.509 certificate that decrypts a **`afkey_evp0`** segment.

Passphrase can be specified two ways:

```
export AFFLIB_PASSPHRASE='mypassphrase'
afinfo file://:mypassphrase@/filename.aff
```

Calling code is unchanged!

AFFLIB encryption example.

```
$ export AFFLIB_PASSPHRASE='password'
$ ./demo
$ afinfo file.aff
file.aff is a AFF file
file.aff: has encrypted segments

file.aff
                                        data
Segment                   arg          length    data
=======               =========     ========    ====
badflag                     0           512   BAD SECTOR..2w..a.....A. ;...
badsectors                  2             8   = 0 (64-bit value)
afflib_version              0             7   "3.5.8"
creator                     0             5   a.out
aff_file_type               0             3   AFF
pagesize             16777216             0
page0                      51             4   ....
imagesize                   2             8   = 65536 (64-bit value)
Bold indicates segments that were decrypted.

Total segments:           9   (9 real)
  Page  segments:         1
  Hash  segments:         0
  Signature segments:     0
  Null segments:          0
```

Without the passphrase, decryption is not possible.

```
$ unset AFFLIB_PASSPHRASE
$ afinfo -a file.aff
file.aff is a AFF file
file.aff: has encrypted segments

Segment                         arg      length    data
=======                   =========    ========    ====
badflag                           0         512   BAD SECTOR..2w..a.....A. ;...+
badsectors                        2           8   = 0 (64-bit value)
afflib_version                    0           7   "3.5.8"
creator                           0           5   a.out
aff_file_type                     0           3   AFF
affkey_aes256                     0          52   ........_.....4>.Nf..q..N..d.
pagesize/aes256            16777216           0
page0/aes256                     51          20   ....dswS.K...NL+....
imagesize/aes256                  2          24   +Y6..3f.......n.........

Total segments:                 9   (9 real)
  Encrypted segments:           3
  Page  segments:               0
  Hash  segments:               0
  Signature segments:           0
  Null segments:                0
  Empty segments:               0

Total data bytes in segments: 631
Total space in file dedicated to segment names: 107
```

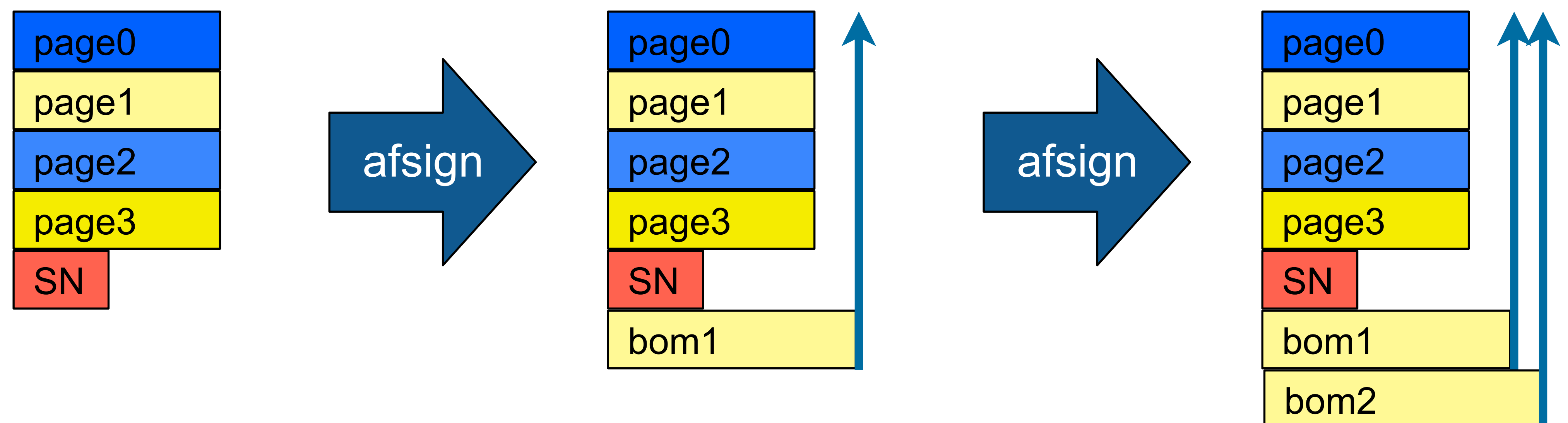
AFFLIBv3 also adds digital signatures and parity pages.

Signatures are as signed SHA256 values.

- Each segment's SHA256 is calculated.
- SHA256 values are signed using OpenSSL's EVP_Sign functions.

Signatures can be stored:

- In individual signature segments.
- In a new *Bill Of Materials (BOM)* segment.



- Multiple signatures can provide for chain-of-custody.
- afsign can also create a "parity page" for RAID-like reconstruction.

AFFLIBv3 status

AFFLIBv3 is in use today for research and education.

- Integrated with SleuthKit.

AFFLIB tools - A set of utilities for manipulating disk images.

- afcat — outputs an AFF file to stdout as a raw file
- afcopy & afconvert — segment-by-segment copying and verification (optional encryption)
- afinfo — prints details about the segments
- afrecover & affix — recovery of data within a corrupted AFF file
- afsign — signature tool
- afverify — verifies signatures
- afcompare — compares two disk images
- afcrypto — encrypt or decrypt a disk image in place
- afdiskprint — generates an XML-based "diskprint" for fast image comparison.
- affuse — allows AFF images to be "mounted" as raw files on Linux.
- afsegemnt — view or modify an individual segment

AFFLIBv3: strengths and weaknesses

Strengths:

- Single archive for storing all of the data and metadata.
- Strong data integrity
- Compact archiving format (16MB segment size, optional LZMA)

Weaknesses:

- Performance.
  - *16MB page size is problematic for some disk images due to MFT fragmentation.*
  - *Caching is only solution at the present:*

```
export AFFLIB_CACHE_PAGES=24          # Dedicates 24*16=384MB to cache
export AFFLIB_CACHE_PAGES=64          # Dedicates 64*16=1GB to cache
```

- Only one disk image per file
  - *Problem for lots of small devices*
- No way to package "logical" files
  - *e.g. FILE.L01*

NPS PRAESTANTIA PER SCIENTIAM

AFF4

AFF4 is designed to overcome AFF3's limitations

AFF4 is a collaborative effort between:

- Michael Cohen (Australian Federal Police; PyFlag)
- Simson Garfinkel (NPS; AFF)
- Bradly Schatz (Director of Schatz Forensic)

Why AFF4?

- Overcome AFF3 performance limitations.
- Need to store more kinds of *structured information* inside the evidence file.
- Unified data model and naming scheme.

Changes from AFF3:

- AFF container is now a ZIP64 file.
- 16MB pages are replaced with two-level Chunk/Bevy model
- libaff4 library in C; most tools written in Python.

NPS PRAESTANTIA PER SCIENTIAM

AFF4 concepts

Information model

- Abstract metadata – exists independent of the file's physical representation

Data model

- Concrete - How the information is represented on disk.

Information Model is based on RDF

Information is represented as statements about subjects.

- Statements have a subject, predicate and value:

```
aff4://1234 is_a “hard disk”
aff4://1234 aff4:size 1E7
```

- Values can be encoded using specialized “data_types."
- Meanings are precise. (They are not just a freely interpreted string.)

```
aff4://1234 aff4:acquired "2010-02-11T13:00:25+00:00"^^xsd:dateTime
```

A group of statements is called a Graph

The Data Model is the physical manifestation of the abstract information model.

Graphs are serialized using RDF serializations

- (e.g. Turtle, XMLRDF etc).

Basic types of AFF4 objects:

- Volumes – store segments within them. Segments are atomic (indivisible) blobs of data.
- Streams – Data objects which can be opened for reading or writing (e.g. segments, images, maps)
- Graphs – Collections of RDF statements – can be written to volumes.

All AFF4 objects are universally referenced to through a unique URL.

Like AFF3, AFF4 objects can be stored in multiple containers.

- AFF4 calls these "Volumes."
- A Volume can be a ZIP64 file, a database, or a collection of files in a directory.

AFF4 ZIP Volumes are AFF4's default volume format.

Uses ZIP64 standard for large file support

- Can be opened by any tool that supports ZIP files…
- … but data segments require special interpretation.

ZIP format is robust.

- There is a growing number of tools to recover corrupt ZIP files.
- Clear distinction between data content and data integrity.

ZIP format is malleable

- Can join / split volumes at any time
- Archive members have a universally unique name – it does not matter where they are stored.

We do not use ZIP64 encryption and signing.

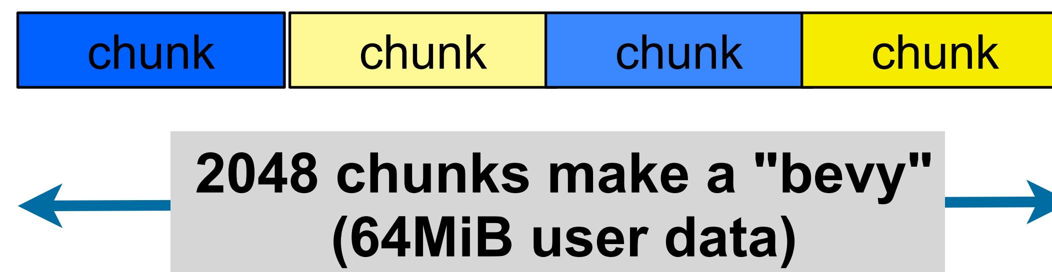
- We implemented our own.

NPS PRAESTANTIA PER SCIENTIAM

AFF4 Image Stream is used for storing seekable, contiguous, compressed data.

Data model is similar to EnCase E01:

- Data is split into chunks (32kb by default)
- Chunks are compressed and written into bevies back to back
  - *— 2048 chunks per bevy by default*



Information model

- Bevy indexes are stored in the aff4:index predicate
- Size is stored in aff4:size predicate
- Typically the information model will be stored in a graph within the volume.

Map streams are a collection of linear byte ranges from other streams.

Every byte in the map stream is taken from an offset of some other stream.

Conceptually maps are an array of points:

- Map offset, Target offset, Target name
- Offsets not in the array are interpolated

Maps are stored in the aff4:map predicate

- Can be encoded using a number of encoders for efficiency (e.g. inline, binary, text)

Map streams can be used for:

- Re-assembling RAID and LVM devices.
- Identifying files within a disk image — useful for zero-copy carving.
- Hash-based imaging — don't stuff archive with objects already in the corpus.
- TCP/IP stream reassembly — Create a map stream from TCP payloads.

NPS PRAESTANTIA PER SCIENTIAM

libaff4 — our implementation of the AFF4 format.

Designed to test ideas and evolve the format by using it.

- Flexible – can combine all types of AFF4 objects together
- Python bindings automatically generated from C source code.
  - *Easy to keep in sync with C library*
  - *C library is very fast; Python bindings make development easy.*
- Multithreaded
- Easy to use

Status:

- API still in flux
- Information on the ForensicsWiki at:
  - *http://www.forensicswiki.org/wiki/AFF4*
  - *http://www.forensicswiki.org/wiki/LibAFF4*
- Download LibAFF4 from:
  - *hg clone https://aff4.googlecode.com/hg/ aff4*

Digital Forensics XML

Digital Forensics XML (DFXML) is a tool for describing file system and file *metadata*.

Today most forensic tools report metadata in human-readable form.

- Location of partitions.
- Location of a file.
- File owner, MAC times, etc.
- Microsoft Office permissions.

This leads to problems:

- Each tool processing a disk image must re-interpret the file system.
- One tool cannot be easily validated against another.

DFXML allows tools to interoperate.

NPS
PRAESTANTIA PER SCIENTIAM

Currently DFXML has four kinds of XML tags.

Per-Image tags

```
<fiwalk> — outer tag
<fiwalk_version>0.4</fiwalk_version>
<Start_time>Mon Oct 13 19:12:09 2008</Start_time>
<Imagefile>dosfs.dmg</Imagefile>
<volume offset="26112">
```

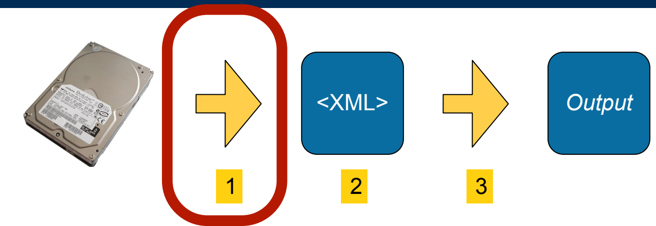
Per <volume> tags:

```
<volume offset="26112">
  <Partition_Offset>26112</Partition_Offset>
  <block_size>512</block_size>
  <ftype>4</ftype>
  <ftype_str>fat16</ftype_str>
  <block_count>60749</block_count>
```

Per <fileobject> tags:

```
<fileobject>
  <filename>DCIM/100CANON/IMG_0001.JPG</filename>
  <filesize>855935</filesize>
  <byte_runs>
    <run file_offset='0' fs_offset='55808' img_offset='81920' len='855935'/>
  </byte_runs>
</fileobject>
```

fiwalk is a tool that produces DFXML files.

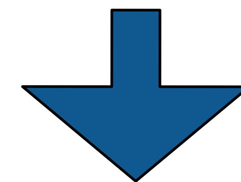


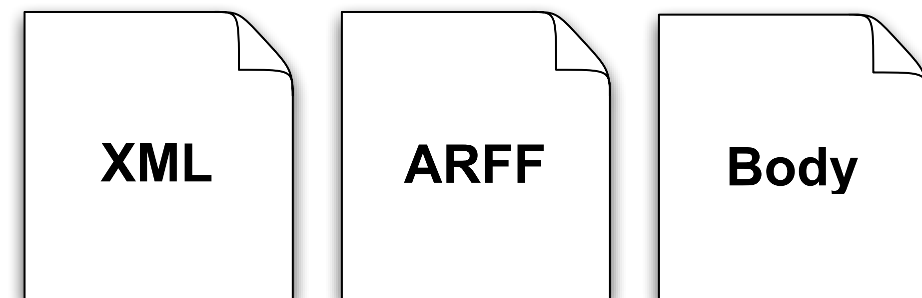
fiwalk is a C++ program built on top of SleuthKit

```
$ fiwalk [options] -X file.xml imagefile
```

Features:

- Finds all partitions & automatically processes each.
- Handles file systems on raw device (partition-less).
- Creates a *single output file* with forensic data data from all.



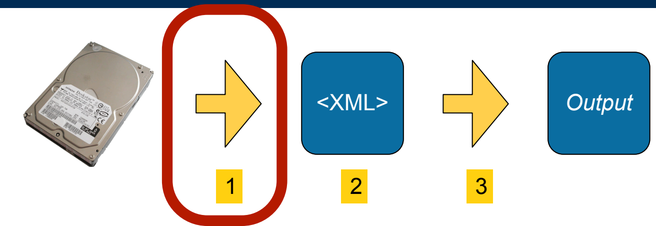


Single program has multiple output formats:

- XML (for automated processing)
- ARFF (for data mining with Weka)
- "walk" format (easy debugging)
- SleuthKit Body File (for legacy timeline tools)
- [CSV (for spreadsheets) ?]

fiwalk has a plugable metadata extraction system.

Configuration file specifies Metadata extractors:



- *Currently the extractor is chosen by the file extension.*

```
*.jpg    dgi      ../plugins/jpeg_extract
*.pdf    dgi      java -classpath plugins.jar Libextract_plugin
*.doc    dgi      java -classpath ../plugins/plugins.jar word_extract
```

- *Plugins are run in a different process for safety.*
- *We have designed a native JVM interface which uses IPC and 1 process.*

Metadata extractors produce name:value pairs on STDOUT

```
Manufacturer: SONY
Model: CYBERSHOT
Orientation: top - left
```

Extracted metadata is automatically incorporated into output.

```
<Manufacturer>SONY</Manufacturer>
<Model>CYBERSHOT</Model>
```

fiwalk's biggest challenge: UTF-8 filenames

Many filesystems allow invalid XML characters in filenames.

- Control Characters
- Invalid Unicode characters (FF) and sequences (EF 32)
- "<" and ">"

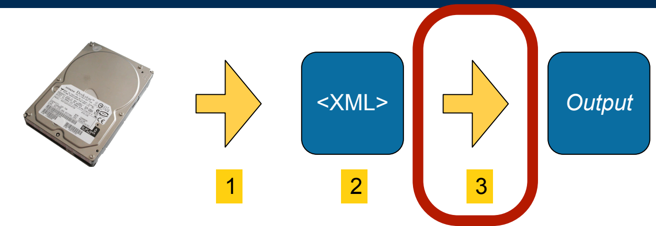
SleuthKit returns UTF-8

- NTFS and HFS *require* valid Unicode in filenames
- Corrupted disks might not have valid Unicode.

Solution: Escaping for both XML and Unicode

- XML escaped — < > etc.
- Control characters are currently turned into "^" by Sleuthkit.
- DEL characters are quoted to \xFF
- Each character is tested for UTF-8; invalid characters escaped (e.g. \xEF \x32)
- "\" is escaped to \x5C

fiwalk.py: a Python module for automated forensics.



Key Features:

- Automatically runs fiwalk with correct options if given a disk image
- Reads XML file if present (faster than regenerating)
- Creates **fileobject** objects.

Multiple interfaces:

- SAX callback interface

```
fiwalk_using_sax(imagefile, xmlfile, flags, callback)
```

  — *Very fast and minimal memory footprint*

- SAX procedural interface

```
objs = fileobjects_using_sax(imagefile, xmlfile, flags)
```

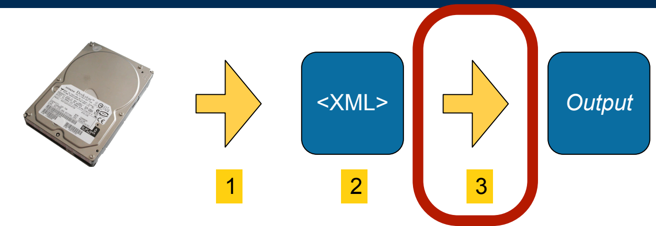
  — *Reasonably fast; returns a list of all file objects with XML in dictionary*

- DOM procedural interface

```
(doc,objs) = fileobjects_using_dom(imagefile, xmlfile, flags)
```

  — *Allows modification of XML that's returned.*

The SAX and DOM interfaces both return fileobjects!



The Python **`fileobject`** class is an easy-to-use abstract class for working with file system data.

Objects belong to one of two subclasses:

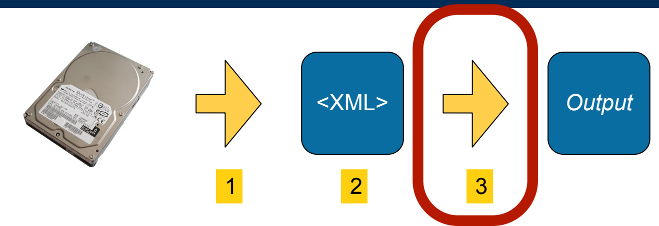
```
fileobject_sax(fileobject)    – for the SAX interface
fileobject_dom(fileobject)    – for the DOM interface
```

Both classes support the same interface:

- *fi.partition()*
- *fi.filename(), fi.ext()*
- *fi.filesize()*
- *fi.ctime(), fi.atime(), fi.crtime(), fi.mtime()*
- *fi.sha1(), fi.md5()*
- *fi.byteruns(), fi.fragments()*
- *fi.content()*

Example: calculate average file size on a disk

Using DOM interface:

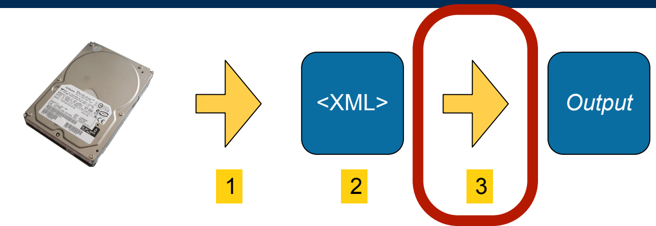



```
import fiwalk

objs = fileobjects_using_sax(imagefile, xmlfile, flags)
print "average file size: ",sum([fi.filesize() for fi in objs]) / len(objs)
```

(For the Python-impaired:)

```
import fiwalk

objs = fileobjects_using_sax(imagefile, xmlfile, flags)
sum_of_sizes = 0
for fi in objs:
    sum_of_sizes += fi.filesize()
print "average file size: ",sum_of_sizes / len(objs)
```

Example: Find and print all the files 15 bytes in length.



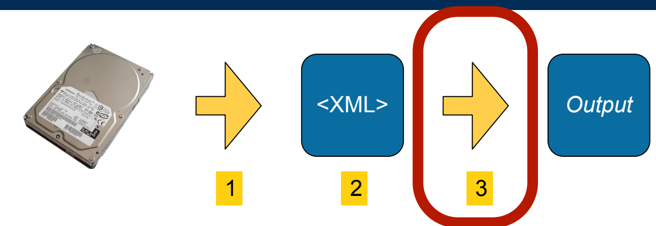
Using DOM interface:

```
import fiwalk

objs = fileobjects_using_sax(imagefile, xmlfile, flags)
for fi in filter(lambda x:x.filesize()==15, objs):
    print fi
```

(For the Python-impaired:)

```
import fiwalk

objs = fileobjects_using_sax(imagefile, xmlfile, flags)
for fi in objs:
    if fi.filesize()==15:
        print fi
```

The fileobject class allows direct access to file data.



byteruns() is an array of “runs.”

```
<byte_runs type='resident'>

  <run file_offset='0' len='65536'
       fs_offset='871588864' img_offset='871621120'/>

  <run file_offset='65536' len='25920'
       fs_offset='871748608' img_offset='871780864'/>

</byte_runs>
```

Becomes:

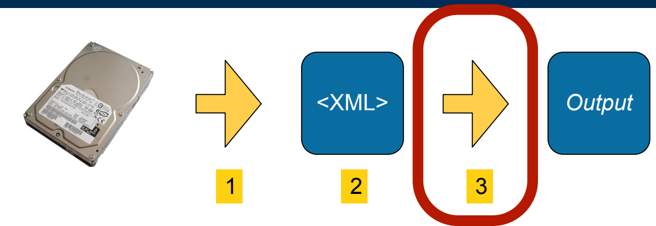
```
[byterun[offset=0; bytes=65536], byterun[offset=65536; bytes=25920]]
```

Each byterun object has:

```
run.start_sector()    – Starting Sector #
run.sector_count()
run.img_offset        - Disk Image offset
run.fs_offset         - File system offset
run.bytes             - number of bytes
run.content()         - content of file
```

The fileobject class allows direct access to file data.

byteruns() returns that array of “runs”
for both the DOM and SAX-based file objects.




```
>>> print fi.byteruns()
[byterun[offset=0; bytes=65536], byterun[offset=65536; bytes=25920]]
```

Accessor Methods:

- fi.contents_for_run(run) — *Returns the bytes from the linked disk image*
- fi.contents() — *Returns all of the contents*
- fi.file_present(imagefile=None) — Validates MD5/SHA1 to see if image has file
- fi.tempfile(calMD5,calcSHA1) — Creates a tempfile, optionally calculating hash

We have several small applications with this framework.

iblkfind.py

- given a disk block in an image, say which files map there.

icarvingtruth.py

- Reports location of carvable files given an earlier XML "map" of the disk image.

idifference.py

- Forensic Disk Differencing

iverify.py

- Reads an image file and XML file; reports which files are actually resident.

imicrosoft_redact.py

- "breaks" a Windows boot disk so that it can be distributed.

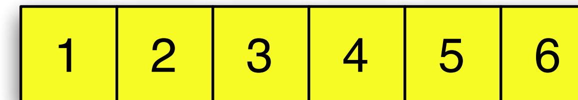
NPS PRAESTANTIA PER SCIENTIAM

iblkfind.py shows how simple it is to build an application.

```
#!/usr/bin/python

import sys,fiwalk

if __name__=="__main__":

    from optparse import OptionParser
    parser = OptionParser()
    parser.usage = '%prog [options] image.iso  s1 [s2 s3 s3 ...]'
    (options,args) = parser.parse_args()

    if len(args)<1:
        parser.print_help()
        sys.exit(1)

    sectors = set([int(n) for n in args[1:]])

    def process(fi):
        for s in sectors:
            if fi.has_sector(s):
                print "%d\t%s" % (s,fi.filename())

    fiwalk.fiwalk_using_sax(imagefile=open(args[0]),callback=process)
```

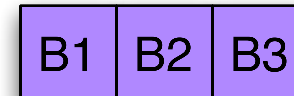
NPS PRAESTANTIA PER SCIENTIAM

frag_find performs hash-based file carving

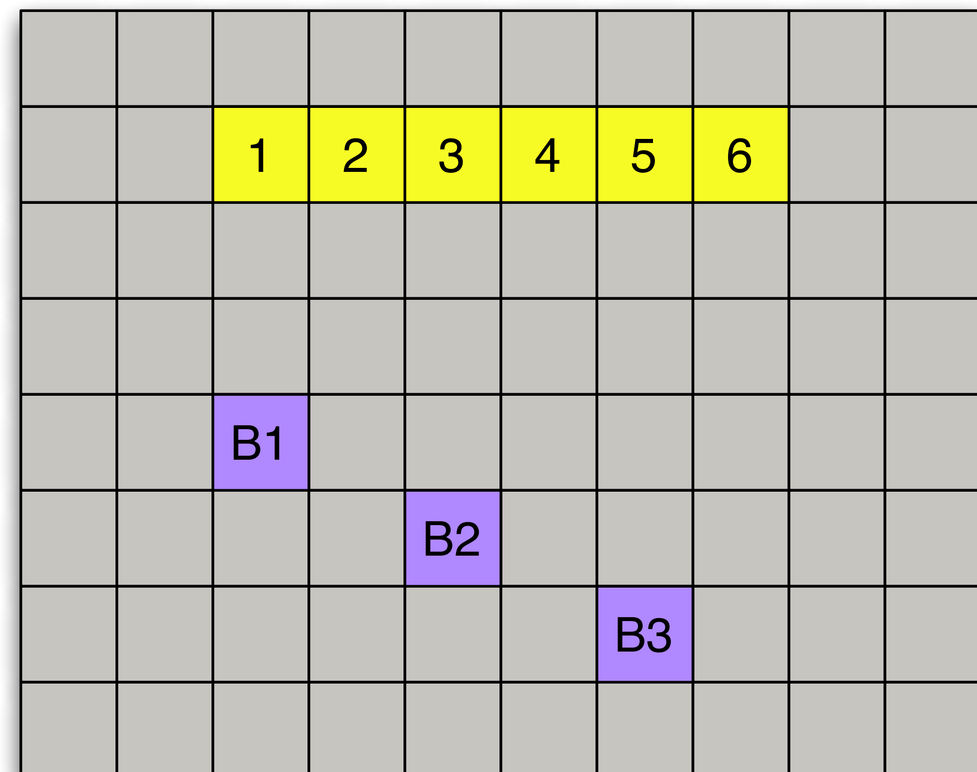


Input:

- 1 or more *Master Files*



- A disk image



Output:

- Digital Forensics XML of where the files are.

```
<fileobject>
  <byte_run>
   …
  </byte_run>
</fileobject>

<fileobject>
  <byte_run>
   …
  </byte_run>
</fileobject>
```

Uses: Exfiltration of sensitive documents; Digital Loss Detection; etc.

bulk_extractor
(sneak preview)

bulk_extractor is a high-speed Named Entity Carver

Input: disk images, memory dumps, network packets, etc.

- No file system interpertation

Output: recognized "Named Entities"

- email addresses
- Credit Card Numbers
- TCP connections
- Parsed URLs (search terms, services, etc.)

Histogram analysis:

- Shows what's important to the subject.

Multi-threaded:

- Turns any CPU-bound task into an I/O-bound task (if you have enough cores)
- Carves a 270GB disk image in 30 minutes.

bulk_extractor sample output: nps-2009-domexusers/email.txt and email_histogram.txt

email feature file:

```
23401051    St@atus.eU
24900678    grafta@bl.com
26735686    grafta@bl.com
32597062    grafta@bl.com
34427974    grafta@bl.com
39265456    domexuser2@gmail.com
39267100    domexuser2@live.com
39269992    domexuser1@gmail.com
39270105    domexuser1@gmail.com
40893040    domexuser2@live.com
40948912    domexuser2@gmail.com
40950441    domexuser2@live.com
42562736    domexuser2@gmail.com
```

Histogram:

```
n=546    domexuser1@gmail.com
n=386    domexuser2@gmail.com
n=331    domexuser3@gmail.com
n=166    domexuser2@live.com
n=140    domexuser2@hotmail.com
n=138    domexuser1@hotmail.com
n=121    domexuser1@live.com
n=94     premium-server@thawte.com
n=57     inet@microsoft.com
n=46     someone@example.com
```

Download your forensic research data today from http://digitalcorpora.org/

In summary:
This talk presented open source tools that you can use.

AFF history and Roadmap

- AFFLIB
- AFF4

API Layer — *interface* to analysis programs.

Schema Layer — *structure* of stored data

Bit-level layer — dictates *how* data is stored

Digital Forensics XML

<fileobject>

- fiwalk
- fiwalk.py

Promote Tools that are available to download NOW!

- frag_find
- bulk_extractor