Tool Assisted File Carving in Hadoop Distributed File System

Frank Klatt

Champlain College

Abstract

Big data systems are becoming more common in many industries, and they present unique challenges to the field of digital forensics. Hadoop Distributed File System (HDFS) is an open source platform built on Apache which provides a modular solution to distributed computation and storage. Forensics investigations on such systems can benefit from having possession of fsimage files, which track metadata including information about where particular data blocks are located throughout the cluster. This paper explains the research and process behind the development of a Python script which is intended to assist digital forensics investigators while carving fsimage files in HDFS. The script can automatically seek and carve fsimage files from a given file and validate them against a Hadoop system in conjunction with the Offline Image Viewer tool. The main challenge of this task was working around the variable "string table" structure at the end of the file. The research draws from previously developed file carving techniques, existing file carving tools, and fundamental knowledge of the Hadoop ecosystem to write the script. With this tool, the author hopes to streamline digital forensics investigations dealing with Hadoop Distributed File System.

Table of Contents

Introduction

As the Industrial Age gave way to the Information Age, technological development did not only accelerate, it accelerated exponentially. Moore’s law not only held but was exceeded. Computers for home and business became more readily available and more powerful at the same time. The digital world overtook the analog, and it permeated all facets of human life: Culture, science, economics, and crime. Law enforcement picked up on this, and in a short time, major governments and law enforcement agencies began developing techniques for uncovering evidence of crimes contained within computers.

Digital forensics has developed in parallel with technology as a whole; new techniques have become important, software companies and government organizations have sprouted up to take on the challenge that digital devices present to law enforcement. Examining larger datasets on larger disks demands more efficient software and more powerful hardware.

Today we are on the cusp of a new age: The Age of Big Data. Large technology companies take advantage of cheap digital storage and collect enormous quantities of data about billions of people in an effort to optimize their produced services. Every Google search is recorded in a database with a person’s profile attached to it. Facebook saves every profile post that every user makes for an extended period of time. Cellular companies collect data on every user’s texts and phone calls. Digital forensics experts make use of this data when they can, but companies are continuing to ramp up their data collection into volumes that ordinary digital infrastructure and file systems strains under. Now, companies that process and store huge amounts of data often use distributed file systems to spread the processing and storage across many coordinated systems. These systems are relatively new, and digital forensics has not yet extensively researched or created tools for them as significantly as traditional file systems that have existed for decades.

Hadoop Distributed File System is a system designed to tackle the challenges associated with big data. It spreads storage and computation across many hierarchical systems, implements redundancy and failure tolerance, and provides a command-line interface. Hadoop and other distributed file systems can contain forensically valuable data. Investigators may need to pull various records from providers like Amazon, which employ these types of distributed file systems. These systems can often be mounted by ordinary operating systems, but if critical file system components are corrupted or missing, investigators have to approach them on a lower level.

File carving is an old and well-explored technique in digital forensics used to recover files that an operating system accessing it can no longer see. Deleted and damaged files or file system data are usually to blame. Many tools exist to recover these files from well-understood operating systems like FAT variations, EXT, and NTFS. Most of these tools depend on distinct file markers to identify the starts and ends of files where the file system is damaged and cannot communicate that data. A particular file known as an “fsimage” has a distinct start of file structure, but no distinct end of file structure. This means that most carving tools will not properly carve fsimage files easily. The particular reason is that a data structure exists at the end of the fsimage file called the String Table, which begins with a simple ascii-type string but is followed by a variable number of bytes that are difficult to predict. Further research may prove it is possible to exactly identify or predict the data of the string table, but for now we can work around this obstacle using simplistic Python scripting.

This document will explain a known file carving technique for Hadoop Distributed File System and use understanding of the technique to develop a simplistic program to streamline and automate it.

This tool makes the process of obtaining intact fsimage files a relatively fast and simple process, needing minimal intervention from a human operator to acquire and verify them. The tool itself automatically carves potential fsimage files from raw data, and checks them against the Hadoop infrastructure using the Offline Image Viewer tool to verify whether the carved data is indeed a valid fsimage file. This eliminates a large quantity of guess work and manual hex editor carving that investigators would need to do if they wanted to obtain fsimage files from a system.

Several assumptions are being made about the use case for the script. The first is that the size of the data being passed to the script is small enough for Python to process it. The second assumption is that the investigator using the script has at least general knowledge about where on the original evidence the fsimage file(s) exist, and that he can cut down the evidence to contain the target material using some other technique such as a hex editor.

More specifically, the use case for this script would be in situations where a digital forensics investigator wants to isolate and process fsimage files from a Hadoop cluster. They may want to do this in the case that the cluster has sustained some damage and will no longer boot, or that storage space has been overwritten and can not be parsed by a standard disk viewing option. The script can parse through misordered file markers and come up with potentially valid pairs in cases of unallocated space, where multiple overwrites and file fragmentation can be present.

Methodology

The methodology involved in producing a file carving tool for HDFS will more closely resemble strategies for developing software, rather than conducting a data-driven research study. Although, in some ways, the strategy will be similar.

The first course of action taken before beginning development is data and environment preparation. The research will be employing virtualized machines configured to host a small Hadoop Distributed File System array, as supplied by Professor Ali Hadi. Data preparation will firstly involve the configuration and mapping of the environment. When the environment is fully prepared, we will begin populating it with test data. In this case, a simple collection of royalty-free stock images is enough to generate complex and workable fsimage files. This data set will be included in the Appendix of this report for reference. By virtualizing the environment in Virtual Box, full control over the systems' hardware and software is managed. The hypervisor also allows for the ability to save "snapshots" of all the systems at a particular point in time, and return to that state instantly and completely. Virtual Box also has a raw disk export feature which is important in generating data to test the tool against.

When the testing environment is fully prepared, the development environment will also be prepared. I prefer Notepad++ since it is a light and flexible text editor that will help structure our script as it is written and modified. Additional tools include those native to the Hadoop stack environment, such as nano. The script material will be offloaded so it will be safe and accessible between system snapshots.

When beginning the scripting, process, prior research tends to be just as important as experimentation and experiential learning. For this script, the base function was simply carving a file from binary data. Some basic parts of the script were borrowed from a simple tutorial in making a jpeg carving script, for example. It is important to have a strong understanding of basic Python syntax.

The script in its final form exists for one purpose, and that is to streamline the carving process used to recover fsimage files from damaged hadoop systems.

The objectives are to develop some basic scenarios to test how well the tool works after the code is written down. The script is run against some basic disk images containing fsimage files as well as some involving irrelevant data and erroneous headers.

Script analysis and experiments

The full script can be found unedited below in the Appendix. Here, the script, carvefsimage.py, is broken down into component parts and explained in detail.

```
import sys # import sys for argv
import os # Import operating system interface
import re # Import regular expressions
import binascii

rawinput = sys.argv[1] # System image passes from first
argument of cmd
FSI_SOF = b'\x48\x44\x46\x53\x49\x4D\x47\x31' # Start of file
marker "HDFSIMG1"
FSI_EOF = b'\x53\x54\x52\x49\x4E\x47\x5F\x54\x41\x42\x4C\x45' #
Marker near end of file "STRING_TABLE"

file_obj=open(rawinput,'rb')data=file_obj.read()
file_obj.close()
```

The above code is the first part of the script. It imports several important libraries: sys, which allows the script to use particular system interface functions and argument passing; os for similar purposes, in particular local file manipulation; re for the definition and use of regular expressions for matching hexadecimal bytes; and binascii for some debugging output.

The next few lines down set up a few variables and establish the data the script will be working with as a workable binary format. The line beginning with "rawinput" sets a variable that will be defined on the command line at launch, such that sending `python3 carvefsimage.py "file-input.dd"` would pass the file name "file-input.dd" from the command line argument into the script variable. The next two lines define the two data structures used to define the approximate starting and ending offset for fsimage files. They are ascii strings, where "HDFSIMG1" is the start of an fsimage and "STRING_TABLE" is very close to the end of the file. Moving down, the next two lines simply read the raw binary data from the file indicated in the command line argument into the variable "file_obj". This is the full file as taken from the system.

```
SOF_list=[match.start() for match in re.finditer(re.escape(FSI_SOF),data)] # Generate list
of offsets for SOF
EOF_list=[match.start() for match in re.finditer(re.escape(FSI_EOF),data)] # Generate list
of offsets for EOF
HF_dict={} # dictionary for describing which offsets are headers (0) and footers (1)
off_pairs=[] # working list of valid offset pairs to be used for carving
# seeking=0 # determines which structure type is being hunted next to be added to offpairs.
0=header 1=footer
working_header=-1 # used for building valid pairs

for offset in SOF_list: # put header offsets into dictionary with value 0
    HF_dict[offset]=0
for offset in EOF_list: # put footer offsets into dictionary with value 1
    HF_dict[offset]=1
for offset, type in sorted(HF_dict.items()): # begin looping through dictionary in order of
offset to build true pairs
    if type==0: # is the offset a header?
        working_header=offset # write it to the working header variable
    else: # if it's not a header
```

```
        if working_header!=-1: # if the working_header has a value...
            off_pairs.append([working_header, offset]) # that means we have found a footer
following a header, so write the pair
            working_header=-1 # reset the loop and remove the previous header from working
header

print("{} , {}".format(SOF_list, EOF_list)) # display the lists of starts and ends
print(off_pairs)
```

This next block of code searches through the given file's data for matches of the header and string_table pseudo-footer, error checks it for floating/missing string matches, and then generates a final list of likely offset-pairs to check for valid fsimage files.

The first two lines, beginning with SOF_list and EOF_list respectively, search through the file for start of file signatures and end of file signatures and puts a list of those offers into two separate list variables. The next line instantiates a dictionary which will later be populated by the below for loops, but it will contain information that identifies which offsets belong to headers and footers. The next line down starting with "off_pairs" instantiates another list which will later be populated by error-checked pairs of offsets to check against the system. These pairs are most likely to be valid fsimage files.

The next block of code consists of two"for" loops dealing with the HF_dict dictionary, and essentially it populates the dictionary with every offset identified in the regular expression matches from earlier, with an indicator key marking header offsets with a 0 and footer offsets with a 1.

The below for loop beginning with "for offset" is a rudimentary error checking loop that sorts and reads the dictionary of header and footer offsets so that they are operated on in order of occurrence on the disk, and then runs through each offset in sequence, checking whether it could be part of a valid [header,footer] offset pair. When it finds a potentially valid pair, the pair is appended to the list “off_pairs.” The logic of the loop can be confusing, but in a simplified sense, it tracks what type of structure was identified at the last read offset and then reacts appropriately if it finds an expected structure key at the next dictionary entry.

At the very end of this chunk, the script first prints the two lists of start structures and end structures, and finally the sorted and error-checked list of offset pairs that it will run through to check for valid fsimage files.

```
for pair in off_pairs:
    for i in range(0,50):
        subdata=data[pair[0]:pair[1]+12+i] # for each SOF entry in SOF_list, set subdata as
the data between sof offset and eof offset + 12 bytes (for "string_table") + extra byte per
attempt
        # print("CHECKING 190-192", binascii.hexlify(subdata[190:192]))
        carve_filename="carvedfsimage"+str(i)+"-"+str(pair[0])+"-"+str(pair[1]) # set the
file name to be written
        carve_obj=open(carve_filename,'wb') # open a new file with the set name. write as
bin
        carve_obj.write(subdata) # write the bytes selected as subdata to the file
        carve_obj.close() # finish the file.
        print ("Found a fsimage and carving it to "+carve_filename) # display which file was
written
        # os.system("cp "+carve_filename+" backupcarve") # copy the carve before pushing to
oiv for debug
        if os.system("hdfs oiv -p XML -i "+carve_filename+" -o fsimagetmp.xml") == 0:
            print("File "+carve_filename+" Validated")
            break # fileisvalidated=1
        else:
            os.system("rm "+carve_filename)
            print("\n------------\n")
    os.system("rm fsimagetmp.xml")
```

This final block of code is the most time consuming of the script primarily because it calls the system to run the HDFS OIV utility against multiple possible fsimage files in a brute-force fashion. It runs through each pair of error-checked offsets and then writes a new raw binary data structure between the HDFSIMG1 structure and the STRING_TABLE structure, plus a certain number of bytes captured after the end of STRING_TABLE. The binary data is written to a new file on the system with a name that contains both the number of bytes captured after the STRING_TABLE as well as the offsets of HDFSIMG1 and STRING_TABLE which defined the pair. Then it checks this file against the system by calling the os.system function, which pushes the command `` `hdfs oiv -p XML -i "+carve_filename+" -o fsimagetmp.xml` `` to the system. This is the command which tells HDFS to run the Offline Image Viewer tool against the carved data and to output a temporary XML file to read. This command will throw an error if the perfect number of bytes is not selected or if the file is otherwise not a valid fsimage, and the loop will toss out the file and try again with an extra byte captured on the end, until it reaches the maximum number of loops. If the command returns a zero, this means the OIV utility was successfully able to read the fsimage and it outputted a readable XML file with the pertinent data. At which point, the loop moves on to the next pair of offsets to carve and check against OIV until it fails or passes. Each successful carve is noted in the terminal and the file is left on the system to be collected and examined, but the XML output is erased.

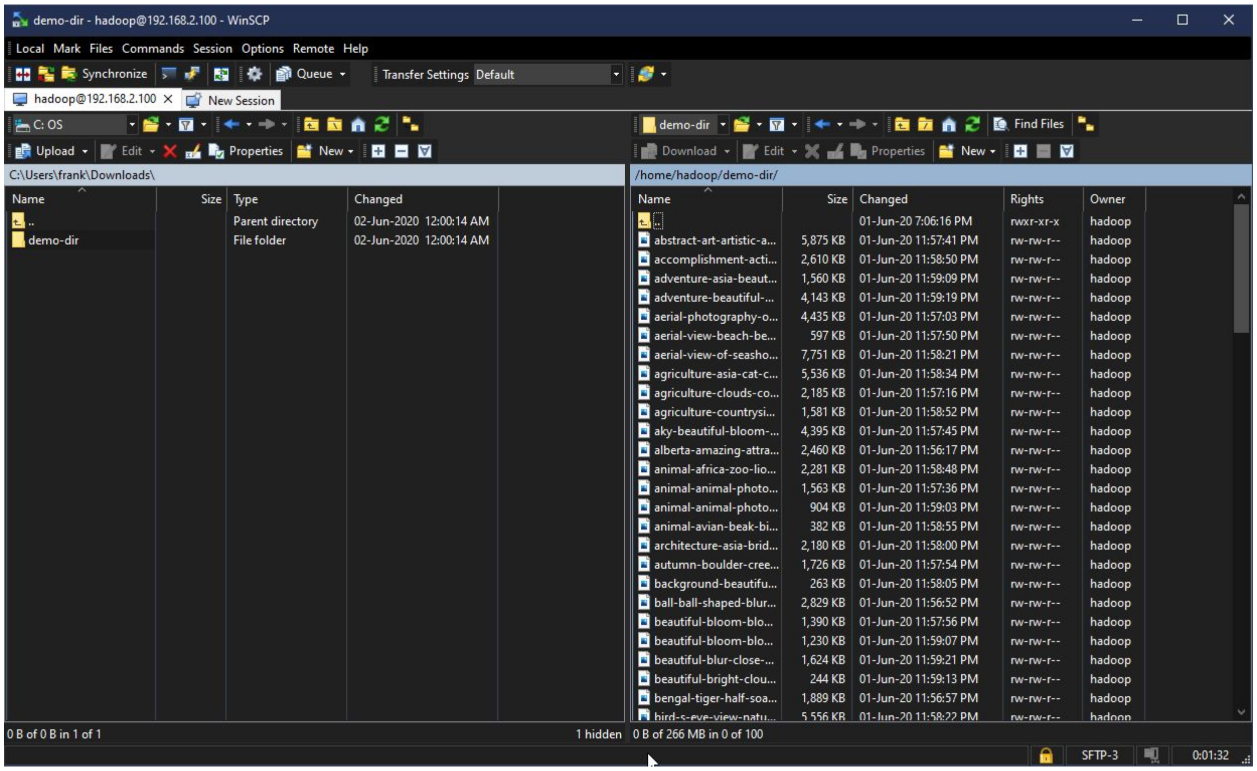
The full code in its entirety will be included in the Appendix below.

Experiments and Results

The script was tested against a limited data set, due to computation and storage constraints. The most complex test was run against a .dd export of a virtual FAT hard disk containing a folder of raw hex files containing fsimage headers. These hex files were carved manually using a hex editor from a .dd image of a Linux system running a Hadoop name node. The carving process involved a string search for HDFSIMG1 and then roughly selecting and exporting hex data in large areas around each header. This means the final sample that the script ran against contains around 9 HDFSIMG1 headers, but these headers were not checked or verified manually for integrity as valid fsimage files. The sample disk and resulting fsimage files carved from it are included in the appendix.

Instructions for the preparation and setup of a virtual Hadoop cluster can be found in the appendix attached to this report. These were produced and released to the public by Ali Hadi and his team.

Below is an example of sample data being uploaded into hadoop using the -put command. This example starts with a folder on Windows containing 100 JPG images; 240 megabytes of data. Uploading this to the cluster with the -put command will generate a substantial fsimage file. WinSCP is the tool used to easily transfer the directory to the Hadoop virtual machine.


demo-dir - hadoop@192.168.2.100 - WinSCP
Local Mark Files Commands Session Options Remote Help
Synchronize Queue Transfer Settings Default
hadoop@192.168.2.100 New Session
C: OS
Upload Edit Properties New
demo-dir Find Files
Download Edit Properties New

C:\Users\frank\Downloads\

Name	Size	Type	Changed
..		Parent directory	02-Jun-2020 12:00:14 AM
demo-dir		File folder	02-Jun-2020 12:00:14 AM

0 B of 0 B in 1 of 1 — 1 hidden

/home/hadoop/demo-dir/

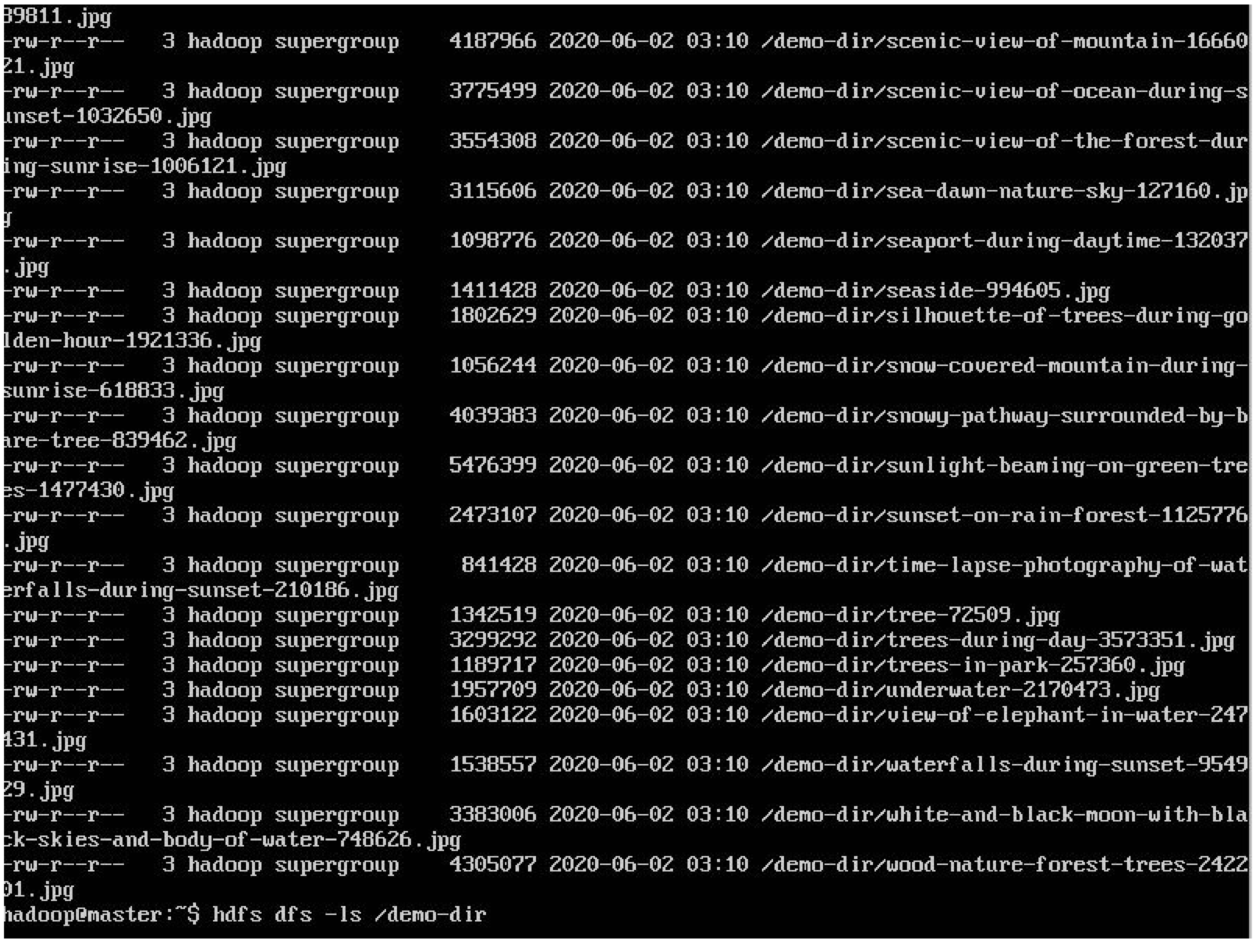
Name	Size	Changed	Rights	Owner
..		01-Jun-20 7:06:16 PM	rwxr-xr-x	hadoop
abstract-art-artistic-a...	5,875 KB	01-Jun-20 11:57:41 PM	rw-rw-r--	hadoop
accomplishment-acti...	2,610 KB	01-Jun-20 11:58:50 PM	rw-rw-r--	hadoop
adventure-asia-beaut...	1,560 KB	01-Jun-20 11:59:09 PM	rw-rw-r--	hadoop
adventure-beautiful-...	4,143 KB	01-Jun-20 11:59:19 PM	rw-rw-r--	hadoop
aerial-photography-o...	4,435 KB	01-Jun-20 11:57:03 PM	rw-rw-r--	hadoop
aerial-view-beach-be...	597 KB	01-Jun-20 11:57:50 PM	rw-rw-r--	hadoop
aerial-view-of-seasho...	7,751 KB	01-Jun-20 11:58:21 PM	rw-rw-r--	hadoop
agriculture-asia-cat-c...	5,536 KB	01-Jun-20 11:58:34 PM	rw-rw-r--	hadoop
agriculture-clouds-co...	2,185 KB	01-Jun-20 11:57:16 PM	rw-rw-r--	hadoop
agriculture-countrysi...	1,581 KB	01-Jun-20 11:58:52 PM	rw-rw-r--	hadoop
aky-beautiful-bloom-...	4,395 KB	01-Jun-20 11:57:45 PM	rw-rw-r--	hadoop
alberta-amazing-attra...	2,460 KB	01-Jun-20 11:56:17 PM	rw-rw-r--	hadoop
animal-africa-zoo-lio...	2,281 KB	01-Jun-20 11:58:48 PM	rw-rw-r--	hadoop
animal-animal-photo...	1,563 KB	01-Jun-20 11:57:36 PM	rw-rw-r--	hadoop
animal-animal-photo...	904 KB	01-Jun-20 11:59:03 PM	rw-rw-r--	hadoop
animal-avian-beak-bi...	382 KB	01-Jun-20 11:58:55 PM	rw-rw-r--	hadoop
architecture-asia-brid...	2,180 KB	01-Jun-20 11:58:00 PM	rw-rw-r--	hadoop
autumn-boulder-cree...	1,726 KB	01-Jun-20 11:57:54 PM	rw-rw-r--	hadoop
background-beautifu...	263 KB	01-Jun-20 11:58:05 PM	rw-rw-r--	hadoop
ball-ball-shaped-blur...	2,829 KB	01-Jun-20 11:56:52 PM	rw-rw-r--	hadoop
beautiful-bloom-blo...	1,390 KB	01-Jun-20 11:57:56 PM	rw-rw-r--	hadoop
beautiful-bloom-blo...	1,230 KB	01-Jun-20 11:59:07 PM	rw-rw-r--	hadoop
beautiful-blur-close-...	1,624 KB	01-Jun-20 11:59:21 PM	rw-rw-r--	hadoop
beautiful-bright-clou...	244 KB	01-Jun-20 11:59:13 PM	rw-rw-r--	hadoop
bengal-tiger-half-soa...	1,889 KB	01-Jun-20 11:56:57 PM	rw-rw-r--	hadoop
bird-s-eye-view-natu...	5,556 KB	01-Jun-20 11:58:22 PM	rw-rw-r--	hadoop

0 B of 266 MB in 0 of 100

SFTP-3 0:01:32

```
hadoop@master:~$ hdfs dfs -put demo-dir /
```

After the put command, a -ls command will show the files on the cluster.

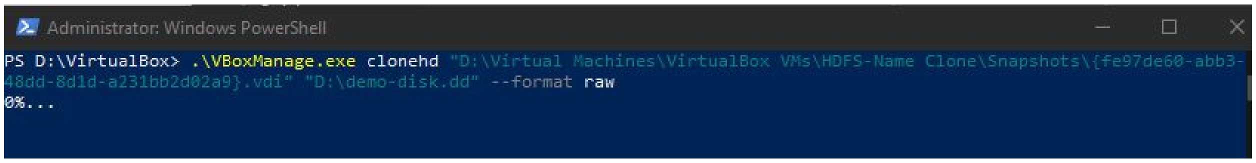


```
39811.jpg
-rw-r--r--   3 hadoop supergroup    4187966 2020-06-02 03:10 /demo-dir/scenic-view-of-mountain-16660
21.jpg
-rw-r--r--   3 hadoop supergroup    3775499 2020-06-02 03:10 /demo-dir/scenic-view-of-ocean-during-s
unset-1032650.jpg
-rw-r--r--   3 hadoop supergroup    3554308 2020-06-02 03:10 /demo-dir/scenic-view-of-the-forest-dur
ing-sunrise-1006121.jpg
-rw-r--r--   3 hadoop supergroup    3115606 2020-06-02 03:10 /demo-dir/sea-dawn-nature-sky-127160.jp
g
-rw-r--r--   3 hadoop supergroup    1098776 2020-06-02 03:10 /demo-dir/seaport-during-daytime-132037
.jpg
-rw-r--r--   3 hadoop supergroup    1411428 2020-06-02 03:10 /demo-dir/seaside-994605.jpg
-rw-r--r--   3 hadoop supergroup    1802629 2020-06-02 03:10 /demo-dir/silhouette-of-trees-during-go
lden-hour-1921336.jpg
-rw-r--r--   3 hadoop supergroup    1056244 2020-06-02 03:10 /demo-dir/snow-covered-mountain-during-
sunrise-618833.jpg
-rw-r--r--   3 hadoop supergroup    4039383 2020-06-02 03:10 /demo-dir/snowy-pathway-surrounded-by-b
are-tree-839462.jpg
-rw-r--r--   3 hadoop supergroup    5476399 2020-06-02 03:10 /demo-dir/sunlight-beaming-on-green-tre
es-1477430.jpg
-rw-r--r--   3 hadoop supergroup    2473107 2020-06-02 03:10 /demo-dir/sunset-on-rain-forest-1125776
.jpg
-rw-r--r--   3 hadoop supergroup     841428 2020-06-02 03:10 /demo-dir/time-lapse-photography-of-wat
erfalls-during-sunset-210186.jpg
-rw-r--r--   3 hadoop supergroup    1342519 2020-06-02 03:10 /demo-dir/tree-72509.jpg
-rw-r--r--   3 hadoop supergroup    3299292 2020-06-02 03:10 /demo-dir/trees-during-day-3573351.jpg
-rw-r--r--   3 hadoop supergroup    1189717 2020-06-02 03:10 /demo-dir/trees-in-park-257360.jpg
-rw-r--r--   3 hadoop supergroup    1957709 2020-06-02 03:10 /demo-dir/underwater-2170473.jpg
-rw-r--r--   3 hadoop supergroup    1603122 2020-06-02 03:10 /demo-dir/view-of-elephant-in-water-247
431.jpg
-rw-r--r--   3 hadoop supergroup    1538557 2020-06-02 03:10 /demo-dir/waterfalls-during-sunset-9549
29.jpg
-rw-r--r--   3 hadoop supergroup    3383006 2020-06-02 03:10 /demo-dir/white-and-black-moon-with-bla
ck-skies-and-body-of-water-748626.jpg
-rw-r--r--   3 hadoop supergroup    4305077 2020-06-02 03:10 /demo-dir/wood-nature-forest-trees-2422
01.jpg
hadoop@master:~$ hdfs dfs -ls /demo-dir
```

Flush the changes to the fsimage using the following commands:

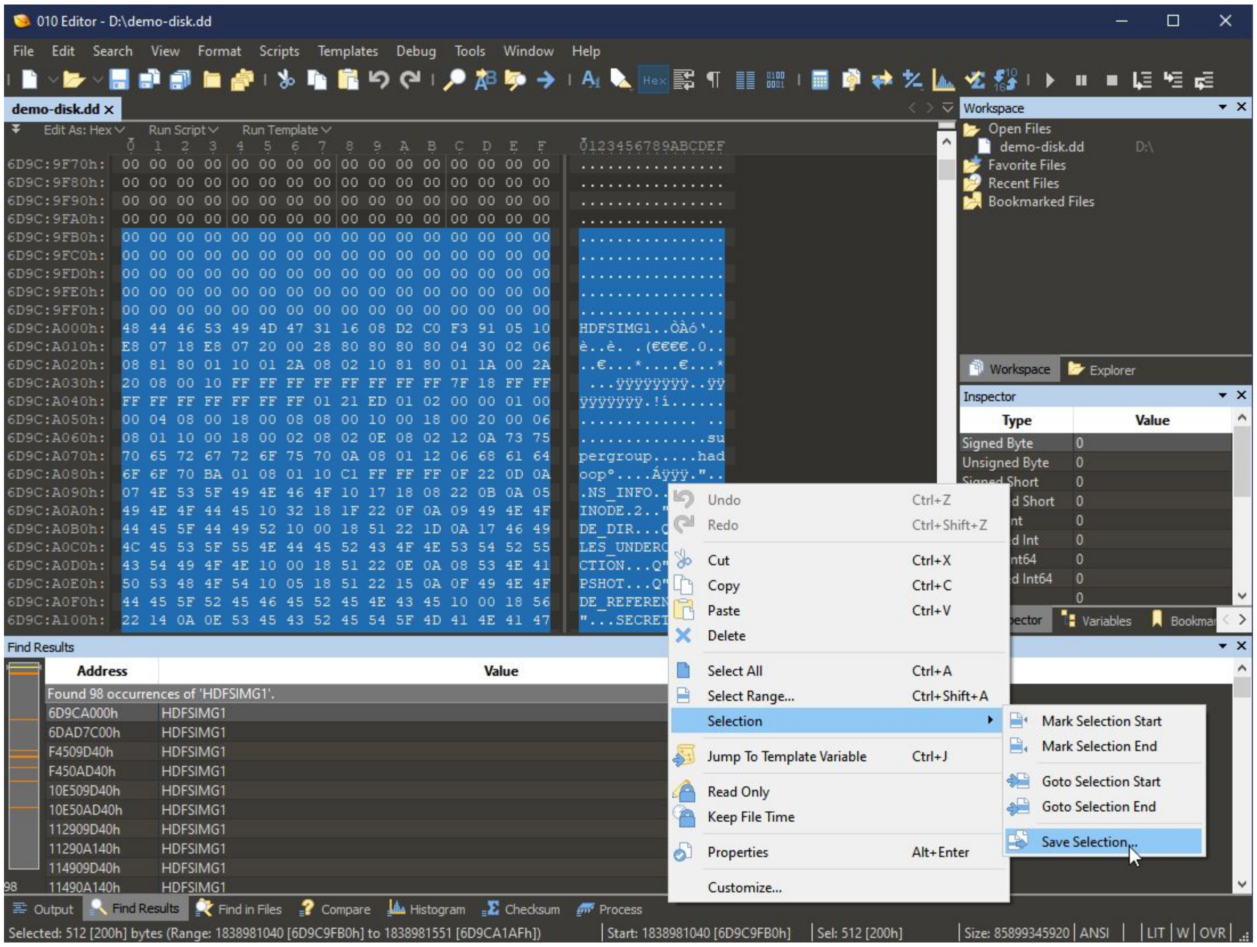
```
$ hdfs dfsadmin -safemode enter
$ hdfs dfsadmin -safemode get # to confirm and ensure it is in safemode
$ hdfs dfsadmin -saveNamespace
$ hdfs dfsadmin -safemode leave
```

Shutdown the cluster and then use command line utility vboxmanage to clone a .dd image of the machine's .vdi file:



```
PS D:\VirtualBox> .\VBoxManage.exe clonehd "D:\Virtual Machines\VirtualBox VMs\HDFS-Name Clone\Snapshots\{fe97de60-abb3-
48dd-8d1d-a231bb2d02a9}.vdi" "D:\demo-disk.dd" --format raw
0%...
```

010 Editor used to roughly hand-carve around HDFSIMG1 headers:



```
            0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
6D9C:9F70h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:9F80h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:9F90h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:9FA0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:9FB0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:9FC0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:9FD0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:9FE0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:9FF0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6D9C:A000h: 48 44 46 53 49 4D 47 31 16 08 D2 C0 F3 91 05 10  HDFSIMG1..ÒÀó'..
6D9C:A010h: E8 07 18 E8 07 20 00 28 80 80 80 80 04 30 02 06  è..è. .(€€€€.0..
6D9C:A020h: 08 81 80 01 10 01 2A 08 02 10 81 80 01 1A 00 2A  ..€...*....€...*
6D9C:A030h: 20 08 00 10 FF FF FF FF FF FF FF FF 7F 18 FF FF   ...ÿÿÿÿÿÿÿÿ..ÿÿ
6D9C:A040h: FF FF FF FF FF FF FF 01 21 ED 01 02 00 00 01 00  ÿÿÿÿÿÿÿ.!í......
6D9C:A050h: 00 04 08 00 18 00 08 08 00 10 00 18 00 20 00 06  ............. ..
6D9C:A060h: 08 01 10 00 18 00 02 08 02 0E 08 02 12 0A 73 75  ..............su
6D9C:A070h: 70 65 72 67 72 6F 75 70 0A 08 01 12 06 68 61 64  pergroup.....had
6D9C:A080h: 6F 6F 70 BA 01 08 01 10 C1 FF FF FF 0F 22 0D 0A  oop°....Áÿÿÿ."..
6D9C:A090h: 07 4E 53 5F 49 4E 46 4F 10 17 18 08 22 0B 0A 05  .NS_INFO..
6D9C:A0A0h: 49 4E 4F 44 45 10 32 18 1F 22 0F 0A 09 49 4E 4F  INODE.2.."
6D9C:A0B0h: 44 45 5F 44 49 52 10 00 18 51 22 1D 0A 17 46 49  DE_DIR...Q
6D9C:A0C0h: 4C 45 53 5F 55 4E 44 45 52 43 4F 4E 53 54 52 55  LES_UNDERC
6D9C:A0D0h: 43 54 49 4F 4E 10 00 18 51 22 0E 0A 08 53 4E 41  CTION...Q"
6D9C:A0E0h: 50 53 48 4F 54 10 05 18 51 22 15 0A 0F 49 4E 4F  PSHOT...Q"
6D9C:A0F0h: 44 45 5F 52 45 46 45 52 45 4E 43 45 10 00 18 56  DE_REFEREN
6D9C:A100h: 22 14 0A 0E 53 45 43 52 45 54 5F 4D 41 4E 41 47  "...SECRET
```

Type	Value
Signed Byte	0
Unsigned Byte	0
Signed Short	0
d Short	0
nt	0
d Int	0
nt64	0
d Int64	0
	0

Find Results

Address	Value
Found 98 occurrences of 'HDFSIMG1'.	
6D9CA000h	HDFSIMG1
6DAD7C00h	HDFSIMG1
F4509D40h	HDFSIMG1
F450AD40h	HDFSIMG1
10E509D40h	HDFSIMG1
10E50AD40h	HDFSIMG1
112909D40h	HDFSIMG1
11290A140h	HDFSIMG1
114909D40h	HDFSIMG1
11490A140h	HDFSIMG1

98

Selected: 512 [200h] bytes (Range: 1838981040 [6D9C9FB0h] to 1838981551 [6D9CA1AFh]) | Start: 1838981040 [6D9C9FB0h] | Sel: 512 [200h] | Size: 85899345920 | ANSI | LIT | W | OVR

Save these cuts to a new VHD (Ideally FAT) and image it with FTKImager:

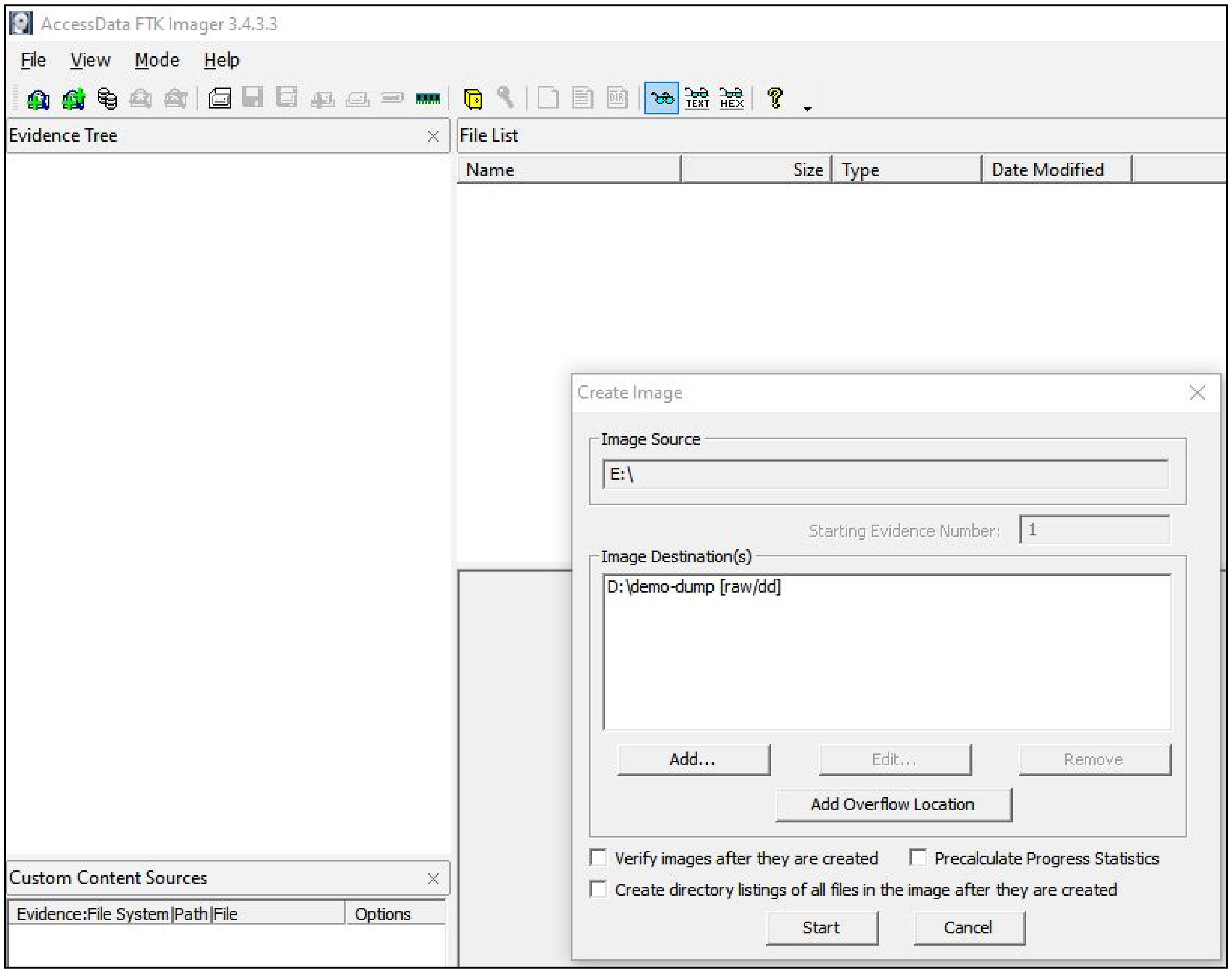


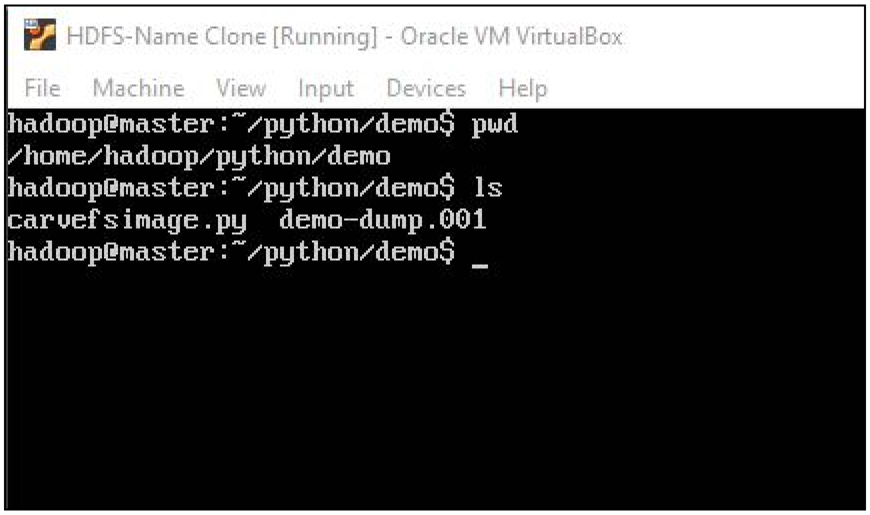
Image Source: E:\

Starting Evidence Number: 1

Image Destination(s): D:\demo-dump [raw/dd]

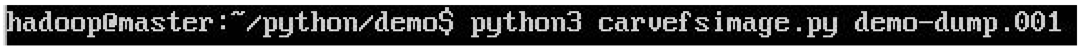
- [ ] Verify images after they are created
- [ ] Precalculate Progress Statistics
- [ ] Create directory listings of all files in the image after they are created

Transfer the .dd to a Hadoop system with OIV working. You don’t need to start the cluster, just a node with the OIV utility.


HDFS-Name Clone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

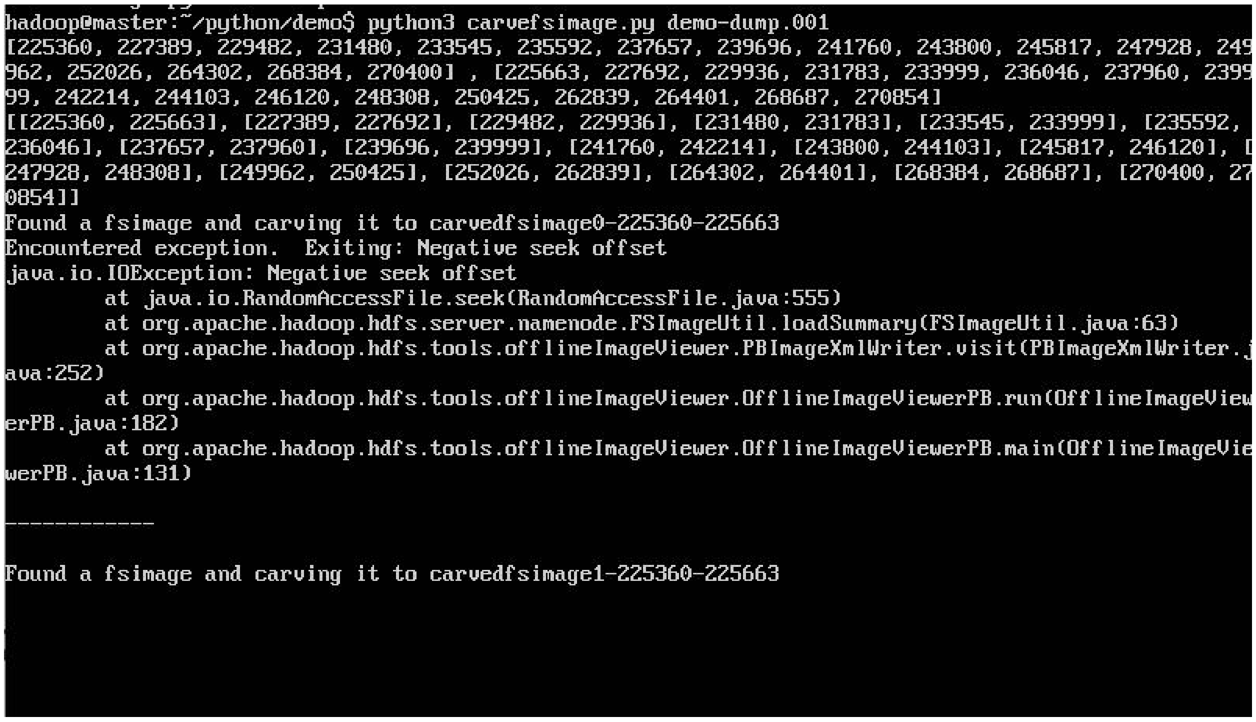
```
hadoop@master:~/python/demo$ pwd
/home/hadoop/python/demo
hadoop@master:~/python/demo$ ls
carvefsimage.py  demo-dump.001
hadoop@master:~/python/demo$ _
```

Run the script over the image:



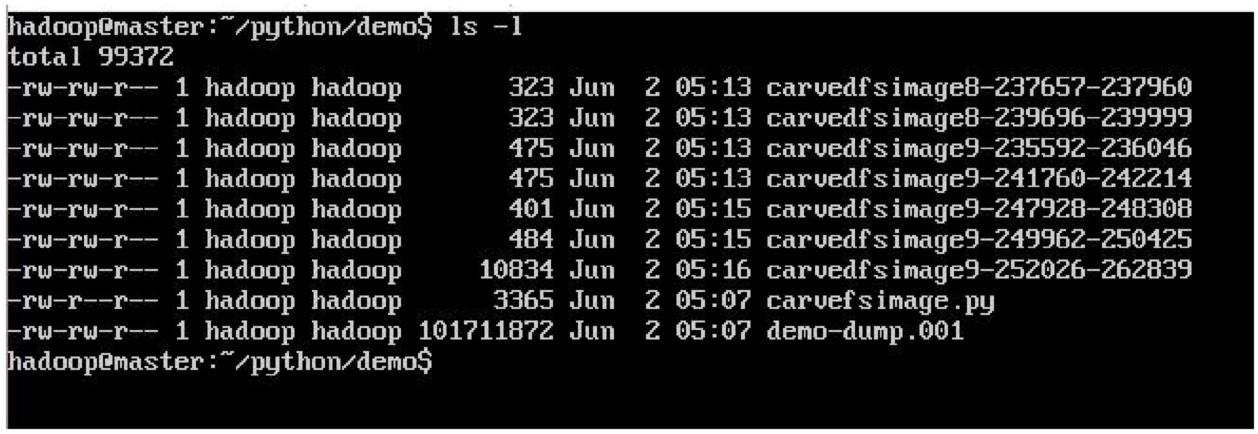
```
hadoop@master:~/python/demo$ python3 carvefsimage.py demo-dump.001
```

You’ll see it start working:



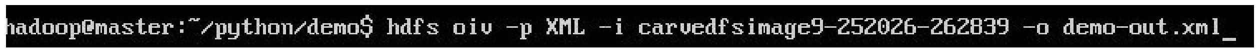
```
hadoop@master:~/python/demo$ python3 carvefsimage.py demo-dump.001
[225360, 227389, 229482, 231480, 233545, 235592, 237657, 239696, 241760, 243800, 245817, 247928, 249
962, 252026, 264302, 268384, 270400] , [225663, 227692, 229936, 231783, 233999, 236046, 237960, 2399
99, 242214, 244103, 246120, 248308, 250425, 262839, 264401, 268687, 270854]
[[225360, 225663], [227389, 227692], [229482, 229936], [231480, 231783], [233545, 233999], [235592,
236046], [237657, 237960], [239696, 239999], [241760, 242214], [243800, 244103], [245817, 246120], [
247928, 248308], [249962, 250425], [252026, 262839], [264302, 264401], [268384, 268687], [270400, 27
0854]]
Found a fsimage and carving it to carvedfsimage0-225360-225663
Encountered exception.  Exiting: Negative seek offset
java.io.IOException: Negative seek offset
        at java.io.RandomAccessFile.seek(RandomAccessFile.java:555)
        at org.apache.hadoop.hdfs.server.namenode.FSImageUtil.loadSummary(FSImageUtil.java:63)
        at org.apache.hadoop.hdfs.tools.offlineImageViewer.PBImageXmlWriter.visit(PBImageXmlWriter.j
ava:252)
        at org.apache.hadoop.hdfs.tools.offlineImageViewer.OfflineImageViewerPB.run(OfflineImageView
erPB.java:182)
        at org.apache.hadoop.hdfs.tools.offlineImageViewer.OfflineImageViewerPB.main(OfflineImageVie
werPB.java:131)

------------

Found a fsimage and carving it to carvedfsimage1-225360-225663
```

The various errors thrown by Java are not to be concerned about. The script will run with full functionality. When it has finished, run ls -l and pay attention to the longest entries.

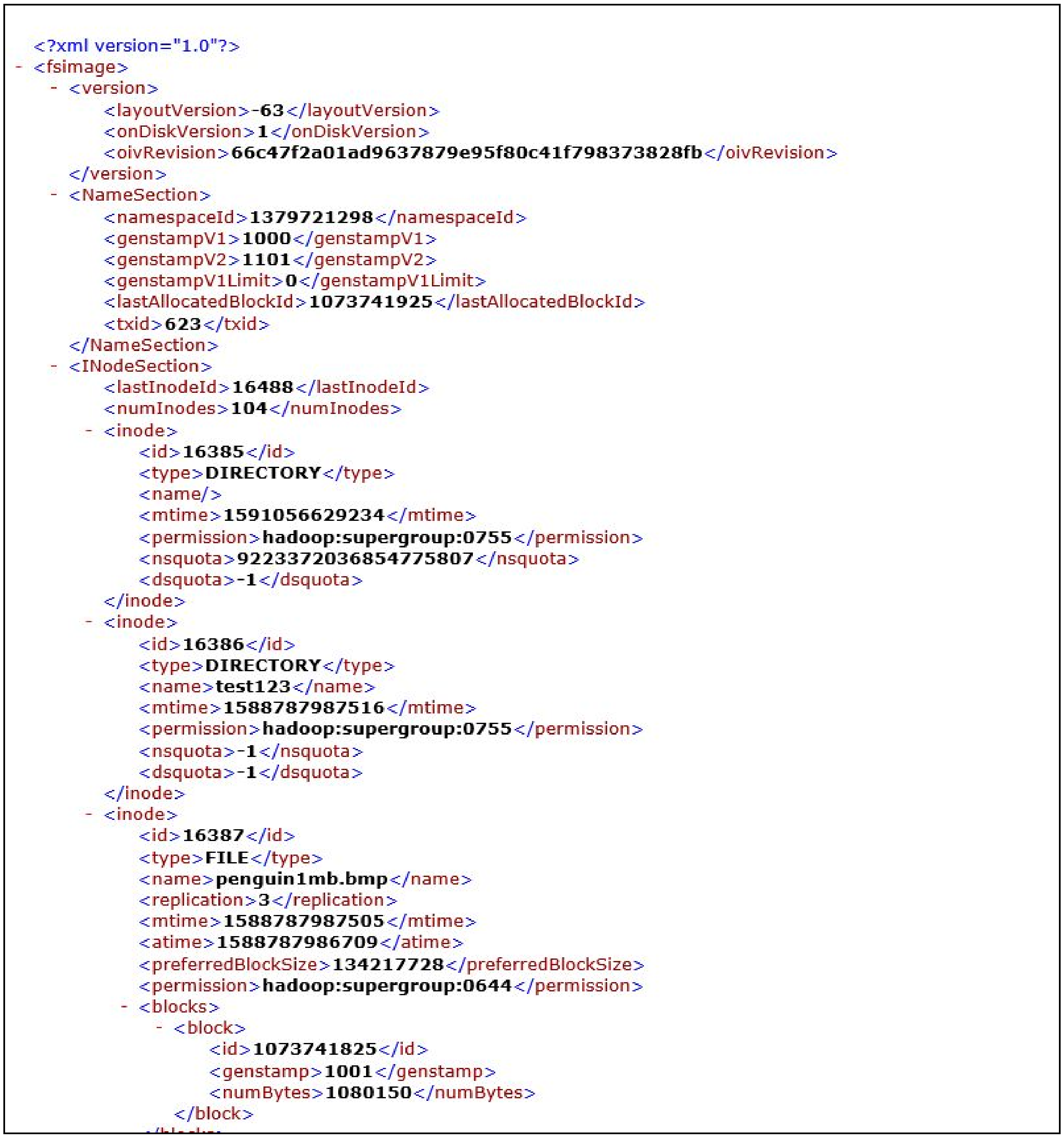


```
hadoop@master:~/python/demo$ ls -l
total 99372
-rw-rw-r-- 1 hadoop hadoop       323 Jun  2 05:13 carvedfsimage8-237657-237960
-rw-rw-r-- 1 hadoop hadoop       323 Jun  2 05:13 carvedfsimage8-239696-239999
-rw-rw-r-- 1 hadoop hadoop       475 Jun  2 05:13 carvedfsimage9-235592-236046
-rw-rw-r-- 1 hadoop hadoop       475 Jun  2 05:13 carvedfsimage9-241760-242214
-rw-rw-r-- 1 hadoop hadoop       401 Jun  2 05:15 carvedfsimage9-247928-248308
-rw-rw-r-- 1 hadoop hadoop       484 Jun  2 05:15 carvedfsimage9-249962-250425
-rw-rw-r-- 1 hadoop hadoop     10834 Jun  2 05:16 carvedfsimage9-252026-262839
-rw-r--r-- 1 hadoop hadoop      3365 Jun  2 05:07 carvefsimage.py
-rw-rw-r-- 1 hadoop hadoop 101711872 Jun  2 05:07 demo-dump.001
hadoop@master:~/python/demo$
```

All of those "carvedfsimage" files are carves from the tool. We can take a look at the XML formatting to see what we captured:



```
hadoop@master:~/python/demo$ hdfs oiv -p XML -i carvedfsimage9-252026-262839 -o demo-out.xml_
```



```
  <?xml version="1.0"?>
- <fsimage>
  - <version>
      <layoutVersion>-63</layoutVersion>
      <onDiskVersion>1</onDiskVersion>
      <oivRevision>66c47f2a01ad9637879e95f80c41f798373828fb</oivRevision>
    </version>
  - <NameSection>
      <namespaceId>1379721298</namespaceId>
      <genstampV1>1000</genstampV1>
      <genstampV2>1101</genstampV2>
      <genstampV1Limit>0</genstampV1Limit>
      <lastAllocatedBlockId>1073741925</lastAllocatedBlockId>
      <txid>623</txid>
    </NameSection>
  - <INodeSection>
      <lastInodeId>16488</lastInodeId>
      <numInodes>104</numInodes>
    - <inode>
        <id>16385</id>
        <type>DIRECTORY</type>
        <name/>
        <mtime>1591056629234</mtime>
        <permission>hadoop:supergroup:0755</permission>
        <nsquota>9223372036854775807</nsquota>
        <dsquota>-1</dsquota>
      </inode>
    - <inode>
        <id>16386</id>
        <type>DIRECTORY</type>
        <name>test123</name>
        <mtime>1588787987516</mtime>
        <permission>hadoop:supergroup:0755</permission>
        <nsquota>-1</nsquota>
        <dsquota>-1</dsquota>
      </inode>
    - <inode>
        <id>16387</id>
        <type>FILE</type>
        <name>penguin1mb.bmp</name>
        <replication>3</replication>
        <mtime>1588787987505</mtime>
        <atime>1588787986709</atime>
        <preferredBlockSize>134217728</preferredBlockSize>
        <permission>hadoop:supergroup:0644</permission>
      - <blocks>
        - <block>
            <id>1073741825</id>
            <genstamp>1001</genstamp>
            <numBytes>1080150</numBytes>
          </block>
```

This looks good! Lots of information can be gained from fsimage files... but that is for another time. Checking the other carvedfsimage files will yield similar results, but since this was the longest carve it will most likely have the data we are looking for. In older production clusters you might find many fsimage files that are much larger than 11kb.

Conclusion

FSImage files can be a valuable resource for digital forensics investigators, and before this python script was written, they were very tedious to acquire from damaged file systems. Because I have automated the trial-and-error process, investigators can carve and verify multiple fsimage files with greater convenience and speed than was previously possible.

Future research could optimize this task further by implementing a more robust interpretation and parsing system for the data following the STRING_TABLE structure. If this data was predicted in some way, it could be directly captured using a regular expression, eliminating the need for a brute force solution such as the one implemented in this paper.

References

Alshammari, Esraa, Ghazi Al-Naymat, Ali Hadi. "A New Technique for File Carving on Hadoop Ecosystem." Amman: IEEE, 11 January 2018. Web. 22 October 2019.

M Khader, A Hadi, G. Al-Naymat, "HDFS file operation fingerprints for forensic investigations[J]", Digital Investigation, vol. 24, pp. 50-61, 2018.

Sremack, Joe. *Big Data Forensics – Learning Hadoop Investigations.* Packt Publishing, 2015. 22 October 2019.

Team, DataFlair. "Hadoop Ecosystem and Their Components - A Complete Tutorial." DataFlair, 26 Apr. 2019, data-flair.training/blogs/hadoop-ecosystem-components/.

Thanekar, Sachin & Subrahmanyam, K. & Bagwan, A.B.. (2016). A Study on Digital Forensics in Hadoop. Indonesian Journal of Electrical Engineering and Computer Science. 9. 8927-8933. 10.11591/ijeecs.v4.i2.pp473-478.

Appendix

Original Script

```
# This script exists to carve fsimage files from corrupted/damaged Hadoop system images.
import sys # import sys for argv
import os # Import operating system interface
import re # Import regular expressions
import binascii

rawinput = sys.argv[1] # System image passes from first argument of cmd
FSI_SOF = b'\x48\x44\x46\x53\x49\x4D\x47\x31' # Start of file marker "HDFSIMG1"
FSI_EOF = b'\x53\x54\x52\x49\x4E\x47\x5F\x54\x41\x42\x4C\x45' # Marker near end of file
"STRING_TABLE"

file_obj=open(rawinput,'rb') # argv[1] should pass the first argument from the run cmd
data=file_obj.read()
# print("CHECKING 190-192", binascii.hexlify(data[190:192]))
file_obj.close()

SOF_list=[match.start() for match in re.finditer(re.escape(FSI_SOF),data)] # Generate list
of offsets for SOF
EOF_list=[match.start() for match in re.finditer(re.escape(FSI_EOF),data)] # Generate list
of offsets for EOF
HF_dict={} # dictionary for describing which offsets are headers (0) and footers (1)
off_pairs=[] # working list of valid offset pairs to be used for carving
# seeking=0 # determines which structure type is being hunted next to be added to offpairs.
0=header 1=footer
working_header=-1 # used for building valid pairs

for offset in SOF_list: # put header offsets into dictionary with value 0
    HF_dict[offset]=0
for offset in EOF_list: # put footer offsets into dictionary with value 1
    HF_dict[offset]=1
for offset, type in sorted(HF_dict.items()): # begin looping through dictionary in order of
offset to build true pairs
    if type==0: # is the offset a header?
        working_header=offset # write it to the working header variable
    else: # if it's not a header
        if working_header!=-1: # if the working_header has a value...
            off_pairs.append([working_header, offset]) # that means we have found a footer
following a header, so write the pair
            working_header=-1 # reset the loop and remove the previous header from working
header

print("{} , {}".format(SOF_list, EOF_list)) # display the lists of starts and ends
print(off_pairs)

for pair in off_pairs:
    for i in range(0,20): # Adjust this value to change maximum attempts per pair. Usually
<20
        subdata=data[pair[0]:pair[1]+12+i] # for each SOF entry in SOF_list, set subdata as
the data between sof offset and eof offset + 12 bytes (for "string_table") + extra byte per
attempt
        carve_filename="carvedfsimage"+str(i)+"-"+str(pair[0])+"-"+str(pair[1]) # set the
file name to be written. This name is descriptive and tells where on the original data it
came from
        carve_obj=open(carve_filename,'wb') # open a new file with the set name. write as
bin
        carve_obj.write(subdata) # write the bytes selected as subdata to the file
        carve_obj.close() # finish the file.
        print ("Found a fsimage and carving it to "+carve_filename) # display which file was
written
        if os.system("hdfs oiv -p XML -i "+carve_filename+" -o fsimagetmp.xml") == 0:
            print("File "+carve_filename+" Validated")
            break
        else:
            os.system("rm "+carve_filename)
            print("\n------------\n")
    os.system("rm fsimagetmp.xml")
```

Additional resources, including demonstration materials:

https://drive.google.com/file/d/154_-vsQYgoZVRa6Xq08Uj1pgy5naDftz/view?usp=sharing

Mirror:

https://github.com/frank-klatt/carvefsimage