Experiment No.2
To use forensics tools of Kali Linux to create mirror image and maintain integrity of source of evidence
Date of Performance:
Date of Submission:

Aim: To use forensics tools of Kali Linux to create mirror image and maintain integrity of source of evidence

Objective: To make use of dd and dcfldd tool create a duplicate copy of the source of evidence and to obtain hash value of the source of evidence.

Theory:

'dd' command in Linux

dd is a command-line utility for Unix and Unix-like operating systems whose primary purpose is to convert and copy files.

- On Unix, device drivers for hardware (such as hard disk drives) and special device files (such as /dev/zero and /dev/random) appear in the file system just like normal files.
- dd can also read and/or write from/to these files, provided that function is implemented in their respective drivers
- As a result, dd can be used for tasks such as backing up the boot sector of a hard drive, and obtaining a fixed amount of random data.
- The dd program can also perform conversions on the data as it is copied, including byte order swapping and conversion to and from the ASCII and EBCDIC text encodings.

1. *To backup the entire harddisk :* To backup an entire copy of a hard disk to another hard disk connected to the same system, execute the dd command as shown. In this dd command example, the UNIX device name of the source hard disk is /dev/hda, and device name of the target hard disk is /dev/hdb.

   # dd if=/dev/sda of=/dev/sdb

   - "if" represents inputfile, and "of" represents output file. So the exact copy of /dev/sda will be available in /dev/sdb.
   - If there are any errors, the above command will fail. If you give the parameter "conv=noerror" then it will continue to copy if there are read errors.
   - Input file and output file should be mentioned very carefully. Just in case, you mention source device in the target and vice versa, you might loss all your data.
   - To copy, hard drive to hard drive using dd command given below, sync option allows you to copy everything using synchronized I/O.

   # dd if=/dev/sda of=/dev/sdb conv=noerror, sync

2. *To backup a Partition* : You can use the device name of a partition in the input file, and in the output either you can specify your target path or image file as shown in the dd command.

   # dd if=/dev/hda1 of=~/partition.img

3. *To create an image of a Hard Disk* : Instead of taking a backup of the hard disk, you can create an image file of the hard disk and save it in other storage devices. There are many advantages of backing up your data to a disk image, one being the ease of use. This method is typically faster than other types of backups, enabling you to quickly restore data following an unexpected catastrophe.It creates the image of a hard disk /dev/hda.

   # dd if=/dev/hda of=~/hdadisk.img

4. *To restore using the Hard Disk Image* : To restore a hard disk with the image file of an another hard disk, the following dd command can be used

   # dd if=hdadisk.img of=/dev/hdb

   The image file hdadisk.img file, is the image of a /dev/hda, so the above command will restore the image of /dev/hda to /dev/hdb.

5. *To create CDROM Backup* : dd command allows you to create an iso file from a source file. So we can insert the CD and enter dd command to create an iso file of a CD content.
   # dd if=/dev/cdrom of=tgsservice.iso bs=2048
   dd command reads one block of input and process it and writes it into an output file. You can specify the block size for input and output file. In the above dd command example, the parameter "bs" specifies the block size for the both the input and output file. So dd uses 2048bytes as a block size in the above command.

'dcfldd' command in Linux

Copy a file, converting and formatting according to the options.
dcfldd was initially developed at Department of Defense Computer Forensics Lab (DCFL). This tool is based on the dd program with the following additional features:

- Hashing on-the-fly: dcfldd can hash the input data as it is being transferred, helping to ensure data integrity.
- Status output: dcfldd can update the user of its progress in terms of the amount of data transferred and how much longer operation will take.
- Flexible disk wipes: dcfldd can be used to wipe disks quickly and with a known pattern if desired.
- Image/wipe verify: dcfldd can verify that a target drive is a bit-for-bit match of the specified input file or pattern.
- Multiple outputs: dcfldd can output to multiple files or disks at the same time.
- Split output: dcfldd can split output to multiple files with more configurability than the split command.
- Piped output and logs: dcfldd can send all its log data and output to commands as well as files natively.
- When dd uses a default block size (bs, ibs, obs) of 512 bytes, dcfldd uses 32768 bytes (32 KiB) which is HUGELY more efficient.
- The following options are present in dcfldd but not in dd: ALGORITHMlog:, errlog, hash, hashconv, hashformat, hashlog, hashlog:, hashwindow, limit, of:, pattern, sizeprobe, split, splitformat, statusinterval, textpattern, totalhashformat, verifylog, verifylog:, vf.
    - **dcfldd** supports the following letters to specify amount of data: k for kilo, M for Mega, G for Giga, T for Tera, P for Peta, E for Exa, Z for Zetta and Y for Yotta. E.g. 10M is equal to 10 MiB. See the Blocks and Bytes section to get other possibilities.

    - bs=**BYTES**

      Force ibs=BYTES and obs=BYTES. Default value is 32768 (32KiB). See Blocks and Bytes section. Warning: the block size will be created in RAM. Make sure you have sufficient amount of free memory.
    - cbs=**BYTES**

      Convert BYTES bytes at a time. (see Blocks and Bytes section)
    - conv=**KEYWORDS**

      Convert the file as per the comma separated keyword list.
    - count=**BLOCKS**

Copy only BLOCKS input blocks. (see Blocks and Bytes section)

- limit=**BYTES**

  Similar to count but using BYTES instead of BLOCKS. (see Blocks and Bytes section)
- ibs=**BYTES**

  Read BYTES bytes at a time. (see Blocks and Bytes section)
- if=**FILE**

  Read from FILE instead of stdin. (see Blocks and Bytes section)
- obs=**BYTES**

  Write BYTES bytes at a time. (see Blocks and Bytes section)
- of=**FILE**

  Write to FILE instead of stdout. NOTE: of=FILE may be used several times to write output to multiple files simultaneously.
- **of:=COMMAND**

  Exec and write output to process COMMAND.
- seek=**BLOCKS**

  Skip BLOCKS obs-sized blocks at start of output. (see Blocks and Bytes section)
- skip=**BLOCKS**

  Skip BLOCKS ibs-sized blocks at start of input. (see Blocks and Bytes section)
- pattern=**HEX**

  Use the specified binary pattern as input. You can use a byte only.
- textpattern=**TEXT**

  Use repeating TEXT as input. You can use a character only.
- errlog=**FILE**

  Send error messages to FILE as well as stderr.
- hash=**NAME**

  Do hash calculation in parallel with the disk reading. Either md5, sha1, sha256, sha384 or sha512 can be used. Default algorithm is md5. To select multiple algorithms to run simultaneously enter the names in a comma separated list.
- hashlog=**FILE**

  Send hash output to FILE instead of stderr. If you are using multiple hash algorithms you can send each to a separate file using the convention ALGORITHMlog=FILE, for example md5log=FILE1, sha1log=FILE2, etc.
- hashwindow=**BYTES**

  Perform a hash on every BYTES amount of data. The partial results will be shown in screen. The default hash is md5 but you can use hash= option to choose other.
- **hashlog:=COMMAND**

  Exec and write hashlog to process COMMAND.
- **ALGORITHMlog:=COMMAND**

  Also works in the same fashion of hashlog:=COMMAND.
- hashconv=**[before|after]**

Perform the hashing before or after the conversions.

- hashformat=**FORMAT**

  Display each hashwindow according to Format the hash format mini-language is described below.
- totalhashformat=**FORMAT**

  Display the total hash value according to Format the hash format mini-language is described below.
- status=**[on|off]**

  Display a continual status message on stderr. Default state is "on".
- statusinterval=**N**

  Update the status message every N blocks. Default value is 256.
- sizeprobe=**[if|of|BYTES]**

  Determine the size of the input or output file or an amount of BYTES for use with status messages. This option gives you a percentage indicator around the sizeprobe value. WARNING: do not use this option against a tape device. (see Blocks and Bytes section)
- split=**BYTES**

  Write every BYTES amount of data to a new file. This operation applies to any of=FILE that follows (split= must be put before of=). (see Blocks and Bytes section)
- splitformat=**[TEXT|MAC|WIN]**

  The file extension format for split operation. You may use "a" for letters and "n" for numbers. If you use annn, an extension started as a000 will be appended; the last possible extension for this format will be z999. splitformat=an will provide a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, b0, b1, b2, b3... If nothing is specified the default format is "nnn". NOTE: the split and splitformat options take effect only for output files (option of=) specified AFTER these options appear in the command line (e.g. split=50M splitformat=annn of=/tmp/test.iso). Likewise, you may specify it several times for different output files within the same command line. You may use as many digits in any combination you would like. E.g. "anaannnaana" would be valid, but a quite insane (see Blocks and Bytes section). Other possible approach is MAC. If "MAC" is used, a suffix dmg and several dmgpart will be appended. In other words, it will generate a partial disk image file, used by the Mac OS X operating system. dmgpart files are usually provided with a corresponding dmg file, which is the master file for the split archive. If dmg is opened in Mac OS X, all dmgpart will be read too. The last option is WIN, which will automatically output file naming of foo.001, foo.002, ..., foo.999, foo.1000, ...
- vf=**FILE**

  Verify that FILE matches the specified input.
- verifylog=**FILE**

  Send verify results to FILE instead of stderr.
- **verifylog:=COMMAND**

  Exec and write verify results to process COMMAND.
- --help

  Display a help page and exit.

- --version

  Output version information and exit.

- **Blocks and Bytes**

  BLOCKS and BYTES may be followed by the following multiplicative suffixes: xM M, c 1, w 2, b 512, kD 1000, k 1024, MD 1,000,000, M 1,048,576, GD 1,000,000,000, G 1,073,741,824, and so on for T, P, E, Z, Y.

- **Keywords**

  Each KEYWORD may be:

- **ascii**

  From EBCDIC to ASCII.
- **ebcdic**

  From ASCII to EBCDIC.
- **ibm**

  From ASCII to alternated EBCDIC.
- **block**

  Pad newline-terminated records with spaces to cbs-size.
- **unblock**

  Replace trailing spaces in cbs-size records with newline.
- **lcase**

  Change upper case to lower case.
- **notrunc**

  Do not truncate the output file.
- **ucase**

  Change lower case to upper case.
- **swab**

  Swap every pair of input bytes.
- **noerror**

  Continue after read errors.
- **sync**

  Pad every input block with NULs to ibs-size. When used with block or unblock, pad with spaces rather than NULs.

- **Format**

The structure of FORMAT may contain any valid text and special variables. The built-in variables are the following format: #variable_name#. To pass FORMAT strings to the program from a command line, it may be
necessary to surround your FORMAT strings with "quotes."
The built-in variables are listed below:

- window_start

The beginning byte offset of the hashwindow.

- window_end

The ending byte offset of the hashwindow.

- block_start

The beginning block (by input blocksize) of the window.

- block_end

The ending block (by input blocksize) of the hash window.

- hash

The hash value.

Algorithm
The name of the hash algorithm.
For example, the default FORMAT for hashformat and totalhashformat are:
hashformat="#window_start# - #window_end#: #hash#" totalhashformat="Total (#algorithm#): #hash#"
The FORMAT structure accepts the following escape codes:
\n Newline
\t Tab
\r Carriage return
\ Insert the '\' character
Insert the '#' character as text, not a variable

Examples

Each following line will create a 100 MiB file containing zeros:

```
$ dcfldd if=/dev/zero of=test bs=1M count=100
$ dcfldd if=/dev/zero of=test bs=100M count=1
$ dcfldd if=/dev/zero of=test bs=50M count=2
$ dcfldd if=/dev/zero of=test limit=100M
```

To create a copy (forensics image) from a disk called /dev/sdb inside a file, using input/output blocks of 4096 bytes (4 KiB) instead of 32 KiB (default):

```
$ dcfldd if=/dev/sdb bs=4096 of=sdb.img
```

As the last example, plus calculating MD5 and SHA256 hashes, putting the results inside sdb.md5 and sdb.sha256. It is very useful for forensics works because the hashes will be processed in real time, avoiding a waste of time to make something as 'dd + md5 + sha256'. Considering that I/O disk is very slow and RAM is very fast, the hashes will be calculated, bit per bit in memory, when the next portion of the disk is read. When all disk was read, all hashes are now ready.

$ dcfldd if=/dev/sdb bs=4096 hash=md5,sha256 md5log=sdb.md5 sha256log=sdb.sha256 of=sdb.img

To validate the image file against the original source:

$ dcfldd if=/dev/sdb vf=sdb.img

Splitting the image in 500 MiB slices, using the default bs value (32 KiB). Note that split= must be put before of= to work:

$ dcfldd if=/dev/sdb split=500M of=sdb.img

At the last example, using from a0000 up to z9999 as suffix for each split file:

$ dcfldd if=/dev/sdb split=500M splitformat=annnn of=sdb.img

Now, dcfldd will work byte per byte (bs=1) and will hop 1056087439 bytes. After this, dcfldd will collect 200000 bytes and write the results to a file called airplane.jpg.

$ dcfldd if=/dev/sda3 bs=1 skip=1056087439 count=200000 of=airplane.jpg

In the last example, the same result could be obtained using "limit" instead of "count". The main difference is that count uses 200000*bs and limit uses 200000 bytes (regardless of the value declared in bs option):

$ dcfldd if=/dev/sda3 bs=1 skip=1056087439 limit=200000 of=airplane.jpg

To write something inside a file, you can use seek. Suppose you want to write a message from a file called message.txt inside a file called target.iso, hopping 200000 bytes from start of file:

$ dcfldd if=message.txt bs=1 seek=200000 of=target.iso

dcfldd also can send a result to be processed by an external command:

$ dcfldd if=text.txt of:="cat | sort -u"

To convert a file from ASCII to EBCDIC:

$ dcfldd if=text.asc conv=ebcdic of=text.ebcdic

To convert a file from EBCDIC to ASCII:

$ dcfldd if=text.ebcdic conv=ascii of=text.asc

Process:

Step 1. Open the Linux terminal

Step 2. Attach the required source of evidence [e.g.. Hard Disk, USB device] to the Computer

Step 3. Use the command dd or dcfldd as mentioned , to do the required task

Conclusion:

Using **dd** and **dcfldd** in Kali Linux enables forensic investigators to create exact bit-by-bit mirror images of storage devices while preserving the integrity of digital evidence. dcfldd enhances dd with features like real-time hashing (e.g., MD5, SHA256), progress tracking, and image verification, making it more suitable for forensic imaging. By generating and validating hash values during duplication, the authenticity and integrity of the evidence are ensured throughout the investigation process