



Department of Electronic & Computer Engineering

Forensic Tool – Phase 2

ET4027 – Computer Forensics

Student Name: Dylan Coffey

Student ID: 18251382

Module Code: ET4027

Date: 11/04/2022

Table of Contents

List of Code Extracts.....	i
List of Figures	ii
1. Requirements.....	1
2. Description of Solution	2
2.1 Main.py.....	2
2.2 Tool.py.....	3
2.3 Partition.py.....	5
2.4 FAT.py.....	6
2.5 NTFS.py.....	8
3. Testing and Results	10
3.1 Provided Image	10
3.2 Created Image (Phase 1 Deliverable).....	12
4. Instructions	14
4.1 Executable	14
4.2 Source Code.....	14
5. Statement of Completion	15
6. Source Code	16
6.1 Main.py.....	16
6.2 Tool.py.....	17
6.3 Partition.py.....	19
6.4 FAT.py.....	20
6.5 NTFS.py.....	22

List of Code Extracts

Code Extract 1: Main.py.....	2
Code Extract 2: Tool.py (read partitions method).....	3
Code Extract 3: Tool.py (get partitions, get bytes, get partition methods).....	4
Code Extract 4: Tool.py (print partitions method)	4
Code Extract 5: Partition.py.....	5

Code Extract 6: FAT.py (get FAT info method)	6
Code Extract 7: FAT.py (get deleted files and print FAT info methods)	7
Code Extract 8: NTFS.py (get NTFS info and get MFT attributes methods)	8
Code Extract 9: NTFS.py (get MFT attribute and get attribute type methods)	9
Code Extract 10: NTFS.py (print NTFS info method).....	9

List of Figures

Figure 1: Information of Provided Image (running tool through a terminal).....	10
Figure 2: Information of Provided Image (running tool by itself).....	11
Figure 3: FTK Imager Report of Created Image	12
Figure 4: Mounted Image	13
Figure 5: Evidence Tree of Created Image.....	13
Figure 6: Partition Information of Created Image	13

1. Requirements

The requirements for the phase 2 assignment were set out in the assignment handout.

These were to write a forensic tool program that can display the partition information, and FAT-16/NTFS volume information of a disk drive image. It can be assumed that the image will have a standard MBR, and the first partition will always be formatted as a FAT-16.

The forensic tool should be able to display the following information:

a) Partition Information:

- Number of partitions
- Start sector of each partition
- Size of each partition
- Type of each partition

b) FAT-16 Volume Information:

- Number of sectors per cluster
- Size of the FAT area
- Size of the root directory
- Sector address of Cluster #2
- First deleted file on the root directory (including its name, start sector, size, and the first 16 characters of its contents)

c) NTFS Volume Information:

- Number of bytes per sector
- Number of sectors per cluster
- Sector address for the \$MFT file record
- Type and length of the first two attributes in the \$MFT file record

The tool should be tested using the provided "*Sample_1.dd*" image file to confirm that it can read the correct information. There aren't any requirements for a user interface and a simple text console input is all that is needed to interact with the tool. Any suitable programming language can be used to program the tool, and an executable or runnable script will need to be produced so the program can be demonstrated in lab.

2. Description of Solution

The programming language that was chosen for this assignment is Python. It was chosen as it is a language that I am familiar with, and it can still produce the required solution at a high-level that is easy to understand and interpret.

The solution can be broken down into five Python files:

- main.py
- tool.py
- partition.py
- fat.py
- ntfs.py

2.1 Main.py

Main.py (see Code Extract 1) is responsible for the running of the forensic tool. The file imports the class Tool from tool.py to read the image file. It checks for a second path argument, which is inputted by running the tool through a terminal. If a second argument is found, the method main() is run, which creates an instance of a Tool object, reads the information of the image file, and then prints it for display.

```
1  from os import path
2      from sys import argv
3
4  from tool import Tool
5
6
7  def main():
8      tool = Tool(image_path)
9      tool.read_partitions()
10     tool.print_partitions()
11
12
13  if __name__ == "__main__":
14     print("ET4027 FORENSIC TOOL - PHASE 2\nNAME = Dylan Coffey\nID = 18251382\n")
15     if len(argv) > 1:
16         image_path = argv[1]
17         main()
18     else:
19         while True:
20             image_path = input("Please drop an image into the terminal or type its path:")
21             image_path = image_path.replace("'", "")
22             if not path.exists(image_path):
23                 print("\nError: image path does not exist!\n")
24             else:
25                 main()
26             input("\nPress <ENTER> to input another image:\n")
27
```

Code Extract 1: Main.py

If the tool is run using the executable by itself, an infinite while loop is started. Inside this loop, it prints the initial input for entering the image file and makes sure that the inputted path exists. If it does exist, the method `main()` is run. After this it waits for the user to press ENTER so they can input another image, where it continues to the start of the while loop again.

2.2 Tool.py

`Tool.py` (see Code Extract 2-4) holds a class that is used to read an image file. The class is passed the path of the image, which is initialised alongside the variables “buffer” to hold the partition entry bytes, and “valid_partitions” to hold the count of how many valid partitions there are. The method `read_partitions()` can be used to open the image file, seek to the first partition offset (0x1BE), read the next 64 bytes (each partition is 16 bytes, and there are 4 partitions) and set the buffer as these bytes.

```
1  from partitions.fat import FAT16
2  from partitions.ntfs import NTFS
3  from partitions.partition import Partition
4
5
6  class Tool:
7      def __init__(self, image_path):
8          self.image_path = image_path
9          self.buffer = bytes(64)
10         self.valid_partitions = 0
11
12     def read_partitions(self):
13         with open(self.image_path, "rb") as image:
14             image.seek(0x1BE)
15             self.buffer = image.read(64)
16
```

Code Extract 2: `Tool.py` (read partitions method)

The method `get_partitions()` is then used to loop through the buffer and return a list of `Partition`, `FAT16`, or `NTFS` objects containing their necessary parameters. For each loop, a “partition_entry” variable is calculated by using an entry offset of 16 which is multiplied by the loops increment value. The method `get_bytes()` is used to loop through the buffer and obtain the bytes of each necessary partition information. The method is passed each information’s size in bytes, as well as their offset plus the current “partition_entry”. Once all the partition information has been obtained, they are passed to the method `get_partition()`, which returns the necessary partition object.

The method `get_partitions()` checks the partition type to determine which object needs to be returned. In the case of the partition type being a FAT-16 or NTFS, the objects getter methods `get_fat_info()` or `get_ntfs_info()` are also called to obtain their volume information. At the end of the method `get_partitions()`, the obtained partition type is checked and if it is valid, the variable “valid_partitions” is incremented.

```

17 def get_partitions(self):
18     partitions = []
19     for i in range(0, 4):
20         partition_entry = i * 16
21         partition_type = self.get_bytes(1, partition_entry + 0x04)
22         start_sector = self.get_bytes(4, partition_entry + 0x08)
23         partition_size = self.get_bytes(4, partition_entry + 0x0C)
24         partition = self.get_partition(partition_type, start_sector, partition_size)
25         partitions.append(partition)
26         if partition_type != 0x00:
27             self.valid_partitions += 1
28     return partitions
29
30 def get_bytes(self, size, offset):
31     byte_array = bytearray()
32     for _ in range(size):
33         byte = self.buffer[offset]
34         byte_array.append(byte)
35         offset += 0x01
36     return int.from_bytes(byte_array, "little")
37
38 def get_partition(self, partition_type, start_sector, partition_size):
39     if partition_type == 0x06:
40         fat_16 = FAT16(self.image_path, start_sector, partition_size)
41         fat_16.get_fat_info()
42         return fat_16
43     if partition_type == 0x07:
44         ntfs = NTFS(self.image_path, start_sector, partition_size)
45         ntfs.get_ntfs_info()
46         return ntfs
47     return Partition(partition_type, start_sector, partition_size)
48

```

Code Extract 3: Tool.py (get partitions, get bytes, get partition methods)

Finally, the method `print_partitions()` can be used to loop through each partition obtained from `get_partitions()`, and print their necessary information. In the case of the partition being a FAT16 or NTFS object, their volume information is also printed. After the loop has completed, the total number of valid partitions is displayed.

```

49 def print_partitions(self):
50     partitions = self.get_partitions()
51     for i, partition in enumerate(partitions):
52         print(f"\n===== Partition {i + 1} =====")
53         print(f"Partition Type = {partition.get_partition_type()}")
54         print(f"Start Sector = {partition.get_start_sector()}")
55         print(f"Partition Size (sectors) = {partition.get_partition_size()}")
56         if isinstance(partition, FAT16):
57             partition.print_fat_info()
58         elif isinstance(partition, NTFS):
59             partition.print_ntfs_info()
60     print(f"\nTotal number of valid partitions = {self.valid_partitions}")
61

```

Code Extract 4: Tool.py (print partitions method)

2.3 Partition.py

Partition.py (see Code Extract 5) holds a model class of a partition, containing the necessary parameters that were set out in the requirements:

- partition_type
- start_sector
- partition_size

The class also contains getter methods that are used to retrieve the partitions parameters.

In the case of get_partition_type(), the method incorporates if-elif-else statements to retrieve the partition type as a string, by comparing it with the various hex codes that correspond with their partition type.

```
1 class Partition:
2     def __init__(self, partition_type, start_sector, partition_size):
3         self.partition_type = partition_type
4         self.start_sector = start_sector
5         self.partition_size = partition_size
6
7     def get_partition_type(self):
8         if self.partition_type == 0x00:
9             return "NOT-VALID"
10        elif self.partition_type == 0x01:
11            return "12-BIT FAT"
12        elif self.partition_type == 0x04:
13            return "16-BIT FAT"
14        elif self.partition_type == 0x05:
15            return "EXTENDED MS-DOS PARTITION"
16        elif self.partition_type == 0x06:
17            return "FAT-16"
18        elif self.partition_type == 0x07:
19            return "NTFS"
20        elif self.partition_type == 0x0B:
21            return "FAT-32 (CHS)"
22        elif self.partition_type == 0x0C:
23            return "FAT-32 (LBA)"
24        elif self.partition_type == 0x0E:
25            return "FAT-16 (LBA)"
26        elif self.partition_type == 0x82:
27            return "LINUX SWAP"
28        elif self.partition_type == 0x83:
29            return "LINUX NATIVE"
30        elif self.partition_type == 0x85:
31            return "LINUX EXTENDED"
32        else:
33            return "NOT-DECODED"
34
35    def get_start_sector(self):
36        return self.start_sector
37
38    def get_partition_size(self):
39        return self.partition_size
40
```

Code Extract 5: Partition.py

2.4 FAT.py

FAT.py (see Code Extract 6-7) holds a model class of a FAT-16. The class implements the Partition class, including its parameters and methods. It also contains the necessary parameters that were set out in the requirements:

- sectors_per_cluster
- fat_area_size
- root_dir_size
- cluster_two_sector
- deleted_files

The class contains the method `get_fat_info()`, which is responsible for obtaining the volume information. It opens the image and uses the obtained start sector to seek to the partition boot sector. From there it reads the necessary information, such as the bytes per sector, sectors per cluster, reserved area size, number of FAT copies, etc. Using this information, calculations are made to obtain further information, such as the FAT area size, root directory size, etc.

```
1  from partitions.partition import Partition
2
3
4  class FAT16(Partition):
5      def __init__(self, image_path, start_sector, partition_size):
6          super().__init__(0x06, start_sector, partition_size)
7          self.image_path = image_path
8          self.sectors_per_cluster = None
9          self.fat_area_size = None
10         self.root_dir_size = None
11         self.cluster_two_sector = None
12         self.deleted_files = []
13
14     def get_fat_info(self):
15         with open(self.image_path, "rb") as image:
16             start_address = self.start_sector * 512
17             image.seek(start_address + 0x0B)
18             bytes_per_sector = int.from_bytes(image.read(2), "little")
19             self.sectors_per_cluster = int.from_bytes(image.read(1), "little")
20             reserved_area_size = int.from_bytes(image.read(2), "little")
21             fat_copies = int.from_bytes(image.read(1), "little")
22             max_root_dir_entries = int.from_bytes(image.read(2), "little")
23             image.seek(start_address + 0x16)
24             fat_sector_size = int.from_bytes(image.read(2), "little")
25             self.fat_area_size = fat_copies * fat_sector_size
26             self.root_dir_size = int((max_root_dir_entries * 32) / bytes_per_sector)
27             root_dir_sector = self.start_sector + reserved_area_size + self.fat_area_size
28             self.cluster_two_sector = root_dir_sector + self.root_dir_size
29             self.get_deleted_files(image, root_dir_sector, max_root_dir_entries, bytes_per_sector)
30
```

Code Extract 6: FAT.py (get FAT info method)

Lastly, the method `get_deleted_files()` is called and passed the opened image, root directory sector, maximum number of root directories, and the bytes per sector.

The method `get_deleted_files()` obtains a list of deleted files from the root directory. Using the passed root directory sector and bytes per sector, the address of the first root directory entry is calculated. This is used alongside the maximum number of root directory entries, to loop through each entry in the root directory. It reads each entry file name, and checks if the first byte in the name equals "0xE5". If it does, the file has been deleted, so it continues to obtain its start cluster and size.

It then calculates the cluster sector address (CSA), which is used to read the first 16 characters of the deleted file. The obtained information is then placed into a dictionary, which is appended to the "deleted_files" list. At the end of the method, the root directory entry variable is incremented by 32 (0x20), so the loop can check the next entry.

```

31 def get_deleted_files(self, image, root_dir_sector, max_root_dir_entries, bytes_per_sector):
32     root_dir_entry = root_dir_sector * bytes_per_sector
33     for _ in range(max_root_dir_entries):
34         image.seek(root_dir_entry)
35         file_name = image.read(11)
36         if file_name[0] == 0xE5:
37             file_name = str(file_name, "ansi").replace(" ", "").replace("TXT", ".TXT")
38             image.seek(root_dir_entry + 0x1A)
39             start_cluster = int.from_bytes(image.read(2), "little")
40             file_size = int.from_bytes(image.read(4), "little")
41             cluster_sector = (self.cluster_two_sector + (start_cluster - 2) *
42                             self.sectors_per_cluster) * bytes_per_sector
43             image.seek(cluster_sector)
44             file_content = str(image.read(16), "ansi")
45             deleted_file = {"name": file_name, "start_cluster": start_cluster,
46                             "size": file_size, "content": file_content}
47             self.deleted_files.append(deleted_file)
48             root_dir_entry += 0x20
49
50 def print_fat_info(self):
51     print("\n[Volume Information]")
52     print(f"sectors per cluster = {self.sectors_per_cluster}")
53     print(f"FAT Area Size (sectors) = {self.fat_area_size}")
54     print(f"Root Directory Size (sectors) = {self.root_dir_size}")
55     print(f"Cluster #2 Sector Address = {self.cluster_two_sector}")
56     for i, file in enumerate(self.deleted_files):
57         print(f"\n[Deleted File {i + 1}]")
58         print(f"File Name = {file.get('name')}")
59         print(f"Start Cluster = {file.get('start_cluster')}")
60         print(f"File Size (bytes) = {file.get('size')}")
61         print(f"File Content (first 16 characters):\n~~~~~ Content Start ~~~~~")
62         print(f"      {file.get('content')}\n~~~~~ Content End ~~~~~")
63

```

Code Extract 7: FAT.py (get deleted files and print FAT info methods)

Finally, the method `print_fat_info()` can be used print the necessary information obtained from the method `get_fat_info()`. It also loops through the deleted files list to display their information, including their name, start cluster, size, and the first 16 characters of their contents.

2.5 NTFS.py

NTFS.py (see Code Extract 8-10) holds a model class of a NTFS. The class implements the Partition class, including its parameters and methods. It also contains the necessary parameters that were set out in the requirements:

- bytes_per_sector
- sectors_per_cluster
- mft_sector
- mft_attributes

The class contains the method `get_ntfs_info()`, which retrieves the volume information. It opens the image and seeks to the partition boot sector. From there it reads the necessary information, such as the bytes per sector, sectors per cluster, and logical cluster number. Using this information, the MFT sector is calculated. The method `get_mft_attributes()` is then called and passed the opened image.

```
1 from partitions.partition import Partition
2
3
4 class NTFS(Partition):
5     def __init__(self, image_path, start_sector, partition_size):
6         super().__init__(0x07, start_sector, partition_size)
7         self.image_path = image_path
8         self.bytes_per_sector = None
9         self.sectors_per_cluster = None
10        self.mft_sector = None
11        self.mft_attributes = []
12
13    def get_ntfs_info(self):
14        with open(self.image_path, "rb") as image:
15            start_address = self.start_sector * 512
16            image.seek(start_address + 0x0B)
17            self.bytes_per_sector = int.from_bytes(image.read(2), "little")
18            self.sectors_per_cluster = int.from_bytes(image.read(1), "little")
19            image.seek(start_address + 0x30)
20            logical_cluster = int.from_bytes(image.read(8), "little")
21            self.mft_sector = self.start_sector + (logical_cluster * self.sectors_per_cluster)
22            self.get_mft_attributes(image)
23
24    def get_mft_attributes(self, image):
25        mft_address = self.mft_sector * self.bytes_per_sector
26        image.seek(mft_address + 0x14)
27        first_offset = int.from_bytes(image.read(2), "little")
28        first_attribute = self.get_mft_attribute(image, mft_address + first_offset)
29        second_offset = first_offset + first_attribute.get("length")
30        second_attribute = self.get_mft_attribute(image, mft_address + second_offset)
31        self.mft_attributes = [first_attribute, second_attribute]
32
```

Code Extract 8: NTFS.py (get NTFS info and get MFT attributes methods)

The method `get_mft_attributes()` obtains a list of the first and second MFT attributes. The MFT sector address is calculated using the obtained bytes per sector and MFT sector. It then seeks to this address and reads the offset of the first attribute, which is passed to the method `get_mft_attribute()`.

The method `get_mft_attribute()` obtains the first attribute by seeking to the passed offset, and reading its type and length. The type string is retrieved using the `get_attribute_type()` method, which incorporates if-elif-else statements to compare it with the various values that correspond with their attribute type. After this, the obtained attribute type and length is returned as a dictionary. This entire process is repeated for the second attribute, but its offset is instead calculated by adding the first offset with the length of the first attribute.

```

43 def get_mft_attribute(self, image, offset):
44     image.seek(offset)
45     attribute_type = self.get_attribute_type(int.from_bytes(image.read(4), "little"))
46     attribute_length = int.from_bytes(image.read(4), "little")
47     return {"type": attribute_type, "length": attribute_length}
48
49 @staticmethod
50 def get_attribute_type(attribute_type):
51     if attribute_type == 16:
52         return "$STANDARD_INFORMATION"
53     elif attribute_type == 32:
54         return "$ATTRIBUTE_LIST"
55     elif attribute_type == 48:
56         return "$FILE_NAME"
57     elif attribute_type == 64:
58         return "$OBJECT_ID"
59     elif attribute_type == 80:
60         return "$SECURITY_DESCRIPTOR"
61     elif attribute_type == 96:
62         return "$VOLUME_NAME"
63     elif attribute_type == 122:
64         return "$VOLUME_INFORMATION"
65     elif attribute_type == 128:
66         return "$DATA"
67     elif attribute_type == 144:
68         return "$INDEX_ROOT"
69     elif attribute_type == 160:
70         return "$INDEX_ALLOCATION"
71     elif attribute_type == 176:
72         return "$BITMAP"
73     elif attribute_type == 192:
74         return "$REPARSE_POINT"
75     elif attribute_type == 256:
76         return "$LOGGED_UTILITY_STREAM"
77     else:
78         return "NOT-DEFINED"
79

```

Code Extract 9: NTFS.py (get MFT attribute and get attribute type methods)

Finally, the method `print_ntfs_info()` can be used to print the necessary information obtained from the method `get_ntfs_info()`. It then loops through the MFT attributes list to display the type and length of the first and second MFT attributes.

```

70 def print_ntfs_info(self):
71     print("\n[Volume Information]")
72     print(f"Bytes per sector = {self.bytes_per_sector}")
73     print(f"Sectors per cluster = {self.sectors_per_cluster}")
74     print(f"MFT Sector Address = {self.mft_sector}")
75     for i, attribute in enumerate(self.mft_attributes):
76         print(f"\n[Attribute {i + 1}]")
77         print(f"Type = {attribute.get('type')}")
78         print(f"Length (bytes) = {attribute.get('length')}")
79

```

Code Extract 10: NTFS.py (print NTFS info method)

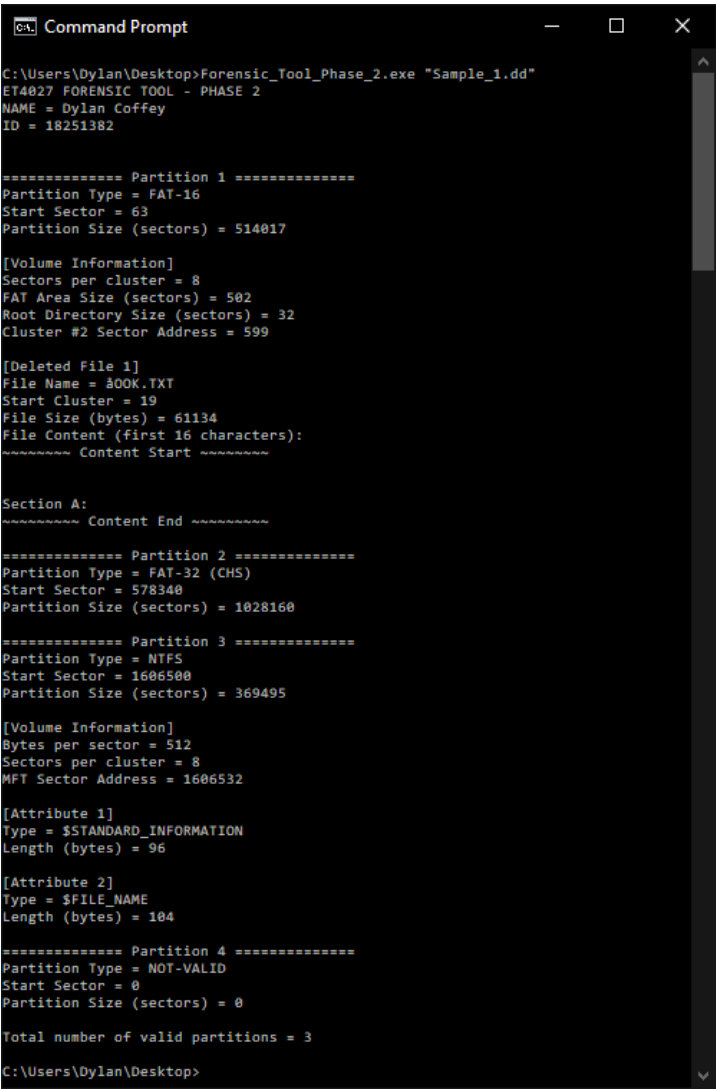
3. Testing and Results

3.1 Provided Image

To test the program during its development a sample image file was provided called "*Sample_1.dd*". The image file consisted of the following three partitions:

- FAT-16
- FAT-32 (CHS)
- NTFS

The program was compiled into an executable and was tested in two ways. The first way was running the program executable through a terminal and passing the image path alongside it. As can be seen in Figure 1, the correct information was displayed.



```
Command Prompt
C:\Users\Dylan\Desktop>Forensic_Tool_Phase_2.exe "Sample_1.dd"
ET4027 FORENSIC TOOL - PHASE 2
NAME = Dylan Coffey
ID = 18251382

===== Partition 1 =====
Partition Type = FAT-16
Start Sector = 63
Partition Size (sectors) = 514017

[Volume Information]
Sectors per cluster = 8
FAT Area Size (sectors) = 502
Root Directory Size (sectors) = 32
Cluster #2 Sector Address = 599

[Deleted File 1]
File Name = 000K.TXT
Start Cluster = 19
File Size (bytes) = 61134
File Content (first 16 characters):
Content Start

Section A:
Content End

===== Partition 2 =====
Partition Type = FAT-32 (CHS)
Start Sector = 578340
Partition Size (sectors) = 1028160

===== Partition 3 =====
Partition Type = NTFS
Start Sector = 1606500
Partition Size (sectors) = 369495

[Volume Information]
Bytes per sector = 512
Sectors per cluster = 8
MFT Sector Address = 1606532

[Attribute 1]
Type = $STANDARD_INFORMATION
Length (bytes) = 96

[Attribute 2]
Type = $FILE_NAME
Length (bytes) = 104

===== Partition 4 =====
Partition Type = NOT-VALID
Start Sector = 0
Partition Size (sectors) = 0

Total number of valid partitions = 3
C:\Users\Dylan\Desktop>
```

Figure 1: Information of Provided Image (running tool through a terminal)

The second way was running the program executable by itself. This provides the user with an input for the path of the image. Entering the images path and pressing <ENTER> displays the required information. As can be seen in Figure 2, the correct information was displayed.

```
C:\Users\Dylan\Desktop\Forensic_Tool_Phase_2.exe
ET4027 FORENSIC TOOL - PHASE 2
NAME = Dylan Coffey
ID = 18251382

Please drop an image into the terminal or type its path:C:\Users\Dylan\Desktop\Sample_1.dd

===== Partition 1 =====
Partition Type = FAT-16
Start Sector = 63
Partition Size (sectors) = 514017

[Volume Information]
Sectors per cluster = 8
FAT Area Size (sectors) = 502
Root Directory Size (sectors) = 32
Cluster #2 Sector Address = 599

[Deleted File 1]
File Name = 000K.TXT
Start Cluster = 19
File Size (bytes) = 61134
File Content (first 16 characters):
Content Start

Section A:
Content End

===== Partition 2 =====
Partition Type = FAT-32 (CHS)
Start Sector = 578340
Partition Size (sectors) = 1028160

===== Partition 3 =====
Partition Type = NTFS
Start Sector = 1606500
Partition Size (sectors) = 369495

[Volume Information]
Bytes per sector = 512
Sectors per cluster = 8
MFT Sector Address = 1606532

[Attribute 1]
Type = $STANDARD_INFORMATION
Length (bytes) = 96

[Attribute 2]
Type = $FILE_NAME
Length (bytes) = 104

===== Partition 4 =====
Partition Type = NOT-VALID
Start Sector = 0
Partition Size (sectors) = 0

Total number of valid partitions = 3
Press <ENTER> to input another image:
```

Figure 2: Information of Provided Image (running tool by itself)

3.2 Created Image (Phase 1 Deliverable)

The following section was only completed during Phase 1 of the assignment, as it was not stated as a deliverable for Phase 2.

To create a new image, the FTK Imager tool was used. A raw (dd) image with 0 fragmentation was created of a USB drive approximately 4GB in size, formatted as a FAT-16 (LBA). A report of the image creation can be seen in Figure 3 below:

```
1 Created By AccessData® FTK® Imager 4.7.1.2
2
3 Case Information:
4 Acquired using: ADI4.7.1.2
5 Case Number: 4027
6 Evidence Number: ET4027
7 Unique description: Created Image
8 Examiner: Dylan Coffey
9 Notes: For testing forensic tool program
10
11 -----
12
13 Information for C:\Users\Dylan\Desktop\FTK Created Image\Created_1:
14
15 Physical Evidentiary Item (Source) Information:
16 [Device Info]
17 Source Type: Physical
18 [Drive Geometry]
19 Cylinders: 478
20 Tracks per Cylinder: 255
21 Sectors per Track: 63
22 Bytes per Sector: 512
23 Sector Count: 7,680,000
24 [Physical Drive Information]
25 Drive Model: USB Device
26 Drive Serial Number: EFX1
27 Drive Interface Type: USB
28 Removable drive: True
29 Source data size: 3750 MB
30 Sector count: 7680000
31 [Computed Hashes]
32 MD5 checksum: e00a313b8b53975d56774a3ff4e950c3
33 SHA1 checksum: 6850ff82flaa6593b334a92926c9a69c92e91689
34
35 Image Information:
36 Acquisition started: Sun Mar 6 18:54:39 2022
37 Acquisition finished: Sun Mar 6 18:58:31 2022
38 Segment list:
39 C:\Users\Dylan\Desktop\FTK Created Image\Created_1.001
40
41 Image Verification Results:
42 Verification started: Sun Mar 6 18:58:34 2022
43 Verification finished: Sun Mar 6 18:59:07 2022
44 MD5 checksum: e00a313b8b53975d56774a3ff4e950c3 : verified
45 SHA1 checksum: 6850ff82flaa6593b334a92926c9a69c92e91689 : verified
46
```

Figure 3: FTK Imager Report of Created Image

After the image had finished creating, it was mounted using the FTK Imager tool to confirm that the image showed the correct files. Word documents containing notes for the current college semester were stored inside the USB drive before the image was created. In Figure 4, the mounted image can be seen within the mapped images menu of the FTK Imager tool.

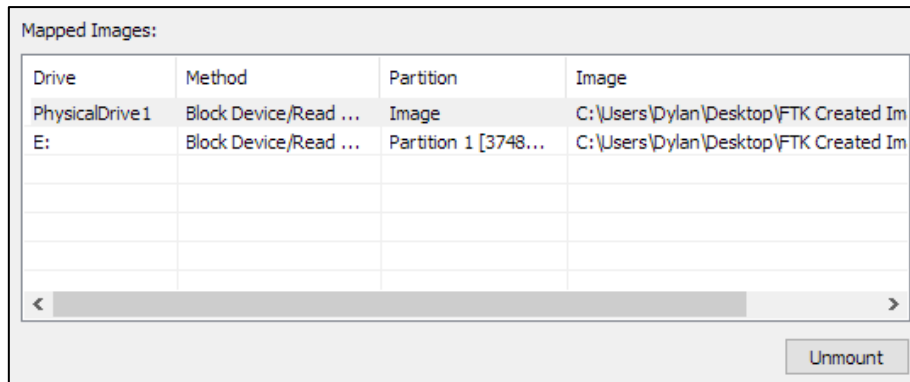


Figure 4: Mounted Image

In Figure 5 below, the created image can be seen added to the evidence tree of the FTK Imager tool. The correct files can be seen inside the FAT-16 (LBA) Partition:

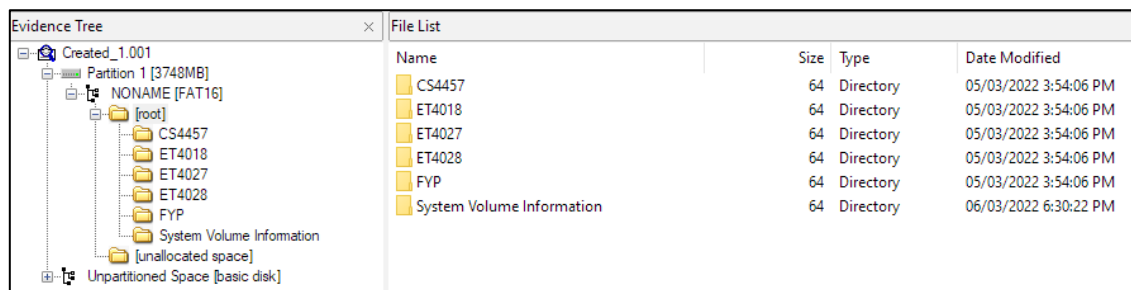


Figure 5: Evidence Tree of Created Image

Finally, the program (Phase 1) was tested against the created image. In Figure 6 below, the FAT-16 (LBA) partition can be seen with its necessary information, including the 3 remaining non-valid partitions:

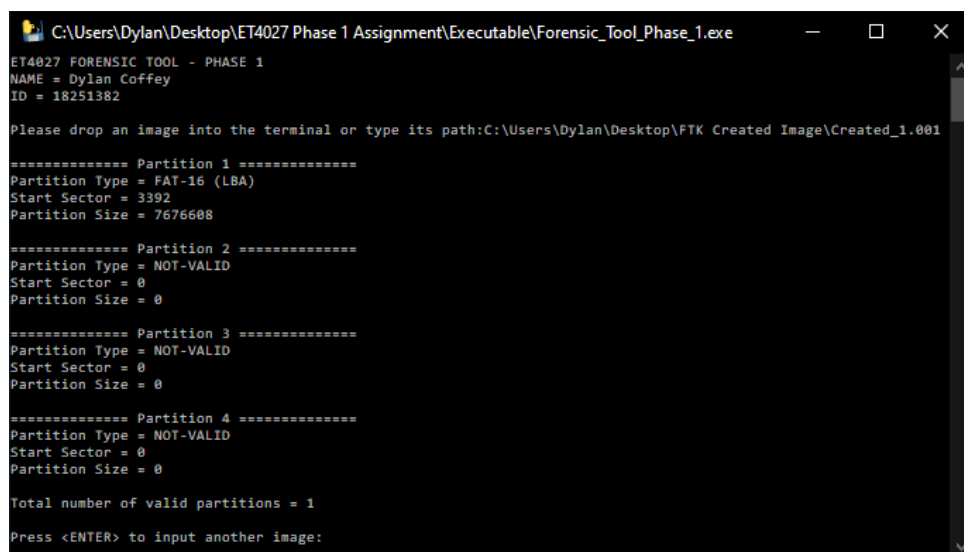


Figure 6: Partition Information of Created Image

4. Instructions

4.1 Executable

To run the forensic tool, it is only necessary to use the compiled executable. This can be run by double clicking it, or through a terminal such as Command Prompt (cmd), Windows PowerShell, or Git Bash. Make sure to include the quotation marks before and after the image path of the command.

- Running through cmd:

Forensic_Tool_Phase_2.exe "image path"

- Running through Windows PowerShell or Git Bash:

./Forensic_Tool_Phase_2.exe "image path"

The executable was created using the plugin PyInstaller, which was installed within the PyCharm IDE. The tool was then compiled through the PyCharm terminal.

- Compile tool through the PyCharm terminal:

pyinstaller main.py --onefile

4.2 Source Code

The source code can also be used to run the tool. However, Python 3.8+ must be installed, and the Python environment path variable must be set on your machine.

- Running source code through a terminal:

python main.py "image path"

An alternative way of running the tool, is to create a new project within PyCharm IDE, and copy the provided source files into it. The tool can then be started by running Main.py.

5. Statement of Completion

I can state that I have completed all the requirements to phase 2 of the forensic tool. The developed program is original work and was developed individually. There weren't any aspects of the assignment that I wasn't able to complete. The completed requirements include:

- Developed a tool that can:
 - Read the image of a disk drive.
 - Display the information of its partitions, including the following:
 - Number of partitions
 - Start sector of each partition
 - Size of each partition
 - Type of each partition
 - Display the volume information of a FAT-16, including the following:
 - Number of sectors per cluster
 - Size of the FAT area
 - Size of the root directory
 - Sector address of Cluster #2
 - First deleted file on the root directory (including its name, start sector, size, and the first 16 characters of its contents)
 - Display the volume information of a NTFS, including the following:
 - Number of bytes per sector
 - Number of sectors per cluster
 - Sector address for the \$MFT file record
 - Type and length of the first two attributes in the \$MFT file record
- Performed testing on the tool using the provided image file called "*Sample_1.dd*".
- Compiled the program into an executable so it can be demonstrated in a lab.

The following Phase 1 requirements were also completed:

- Tested the Phase 1 forensic tool with a created image of a USB drive.
- Provided evidence that the created image can be mounted.

6. Source Code

6.1 Main.py

```
from os import path
from sys import argv

from tool import Tool

def main():
    tool = Tool(image_path)
    tool.read_partitions()
    tool.print_partitions()

if __name__ == "__main__":
    print("ET4027 FORENSIC TOOL - PHASE 2\nNAME = Dylan Coffey\nID = 18251382\n")
    if len(argv) > 1:
        image_path = argv[1]
        main()
    else:
        while True:
            image_path = input("Please drop an image into the terminal or type its path:")
            image_path = image_path.replace("'", "")
            if not path.exists(image_path):
                print("\nError: image path does not exist!\n")
            else:
                main()
                input("\nPress <ENTER> to input another image:\n")
```

6.2 Tool.py

```
from partitions.fat import FAT16
from partitions.ntfs import NTFS
from partitions.partition import Partition

class Tool:
    def __init__(self, image_path):
        self.image_path = image_path
        self.buffer = bytes(64)
        self.valid_partitions = 0

    def read_partitions(self):
        with open(self.image_path, "rb") as image:
            image.seek(0x1BE)
            self.buffer = image.read(64)

    def get_partitions(self):
        partitions = []
        for i in range(0, 4):
            partition_entry = i * 16
            partition_type = self.get_bytes(1, partition_entry + 0x04)
            start_sector = self.get_bytes(4, partition_entry + 0x08)
            partition_size = self.get_bytes(4, partition_entry + 0x0C)
            partition = self.get_partition(partition_type, start_sector, partition_size)
            partitions.append(partition)
            if partition_type != 0x00:
                self.valid_partitions += 1
        return partitions

    def get_bytes(self, size, offset):
        byte_array = bytearray()
        for _ in range(size):
            byte = self.buffer[offset]
            byte_array.append(byte)
            offset += 0x01
        return int.from_bytes(byte_array, "little")

    def get_partition(self, partition_type, start_sector, partition_size):
        if partition_type == 0x06:
            fat_16 = FAT16(self.image_path, start_sector, partition_size)
            fat_16.get_fat_info()
            return fat_16
        if partition_type == 0x07:
            ntfs = NTFS(self.image_path, start_sector, partition_size)
            ntfs.get_ntfs_info()
```

```

        return ntfs
    return Partition(partition_type, start_sector, partition_size)

def print_partitions(self):
    partitions = self.get_partitions()
    for i, partition in enumerate(partitions):
        print(f"\n===== Partition {i + 1} =====")
        print(f"Partition Type = {partition.get_partition_type()}")
        print(f"Start Sector = {partition.get_start_sector()}")
        print(f"Partition Size (sectors) = {partition.get_partition_size()}")
        if isinstance(partition, FAT16):
            partition.print_fat_info()
        elif isinstance(partition, NTFS):
            partition.print_ntfs_info()
    print(f"\nTotal number of valid partitions = {self.valid_partitions}")

```

6.3 Partition.py

```
class Partition:
    def __init__(self, partition_type, start_sector, partition_size):
        self.partition_type = partition_type
        self.start_sector = start_sector
        self.partition_size = partition_size

    def get_partition_type(self):
        if self.partition_type == 0x00:
            return "NOT-VALID"
        elif self.partition_type == 0x01:
            return "12-BIT FAT"
        elif self.partition_type == 0x04:
            return "16-BIT FAT"
        elif self.partition_type == 0x05:
            return "EXTENDED MS-DOS PARTITION"
        elif self.partition_type == 0x06:
            return "FAT-16"
        elif self.partition_type == 0x07:
            return "NTFS"
        elif self.partition_type == 0x0B:
            return "FAT-32 (CHS)"
        elif self.partition_type == 0x0C:
            return "FAT-32 (LBA)"
        elif self.partition_type == 0x0E:
            return "FAT-16 (LBA)"
        elif self.partition_type == 0x82:
            return "LINUX SWAP"
        elif self.partition_type == 0x83:
            return "LINUX NATIVE"
        elif self.partition_type == 0x85:
            return "LINUX EXTENDED"
        else:
            return "NOT-DECODED"

    def get_start_sector(self):
        return self.start_sector

    def get_partition_size(self):
        return self.partition_size
```

6.4 FAT.py

```
from partitions.partition import Partition
```

```
class FAT16(Partition):
```

```
    def __init__(self, image_path, start_sector, partition_size):
```

```
        super().__init__(0x06, start_sector, partition_size)
```

```
        self.image_path = image_path
```

```
        self.sectors_per_cluster = None
```

```
        self.fat_area_size = None
```

```
        self.root_dir_size = None
```

```
        self.cluster_two_sector = None
```

```
        self.deleted_files = []
```

```
    def get_fat_info(self):
```

```
        with open(self.image_path, "rb") as image:
```

```
            start_address = self.start_sector * 512
```

```
            image.seek(start_address + 0x0B)
```

```
            bytes_per_sector = int.from_bytes(image.read(2), "little")
```

```
            self.sectors_per_cluster = int.from_bytes(image.read(1), "little")
```

```
            reserved_area_size = int.from_bytes(image.read(2), "little")
```

```
            fat_copies = int.from_bytes(image.read(1), "little")
```

```
            max_root_dir_entries = int.from_bytes(image.read(2), "little")
```

```
            image.seek(start_address + 0x16)
```

```
            fat_sector_size = int.from_bytes(image.read(2), "little")
```

```
            self.fat_area_size = fat_copies * fat_sector_size
```

```
            self.root_dir_size = int((max_root_dir_entries * 32) / bytes_per_sector)
```

```
            root_dir_sector = self.start_sector + reserved_area_size + self.fat_area_size
```

```
            self.cluster_two_sector = root_dir_sector + self.root_dir_size
```

```
            self.get_deleted_files(image, root_dir_sector, max_root_dir_entries,  
bytes_per_sector)
```

```
    def get_deleted_files(self, image, root_dir_sector, max_root_dir_entries,  
bytes_per_sector):
```

```
        root_dir_entry = root_dir_sector * bytes_per_sector
```

```
        for _ in range(max_root_dir_entries):
```

```
            image.seek(root_dir_entry)
```

```
            file_name = image.read(11)
```

```
            if file_name[0] == 0xE5:
```

```
                file_name = str(file_name, "ansi").replace(" ", "").replace("TXT", ".TXT")
```

```
                image.seek(root_dir_entry + 0x1A)
```

```
                start_cluster = int.from_bytes(image.read(2), "little")
```

```
                file_size = int.from_bytes(image.read(4), "little")
```

```
                cluster_sector = (self.cluster_two_sector + (start_cluster - 2) *
```

```
                    self.sectors_per_cluster) * bytes_per_sector
```

```
                image.seek(cluster_sector)
```

```

        file_content = str(image.read(16), "ansi")
        deleted_file = {"name": file_name, "start_cluster": start_cluster,
                        "size": file_size, "content": file_content}
        self.deleted_files.append(deleted_file)
        root_dir_entry += 0x20

def print_fat_info(self):
    print("\n[Volume Information]")
    print(f"Sectors per cluster = {self.sectors_per_cluster}")
    print(f"FAT Area Size (sectors) = {self.fat_area_size}")
    print(f"Root Directory Size (sectors) = {self.root_dir_size}")
    print(f"Cluster #2 Sector Address = {self.cluster_two_sector}")
    for i, file in enumerate(self.deleted_files):
        print(f"\n[Deleted File {i + 1}]")
        print(f"File Name = {file.get('name')}")
        print(f"Start Cluster = {file.get('start_cluster')}")
        print(f"File Size (bytes) = {file.get('size')}")
        print(f"File Content (first 16 characters):\n~~~~~ Content Start ~~~~~"
              f"\n{file.get('content')}\n~~~~~ Content End ~~~~~")

```


6.5 NTFS.py

```
from partitions.partition import Partition
```

```
class NTFS(Partition):
    def __init__(self, image_path, start_sector, partition_size):
        super().__init__(0x07, start_sector, partition_size)
        self.image_path = image_path
        self.bytes_per_sector = None
        self.sectors_per_cluster = None
        self.mft_sector = None
        self.mft_attributes = []

    def get_ntfs_info(self):
        with open(self.image_path, "rb") as image:
            start_address = self.start_sector * 512
            image.seek(start_address + 0x0B)
            self.bytes_per_sector = int.from_bytes(image.read(2), "little")
            self.sectors_per_cluster = int.from_bytes(image.read(1), "little")
            image.seek(start_address + 0x30)
            logical_cluster = int.from_bytes(image.read(8), "little")
            self.mft_sector = self.start_sector + (logical_cluster * self.sectors_per_cluster)
            self.get_mft_attributes(image)

    def get_mft_attributes(self, image):
        mft_address = self.mft_sector * self.bytes_per_sector
        image.seek(mft_address + 0x14)
        first_offset = int.from_bytes(image.read(2), "little")
        first_attribute = self.get_mft_attribute(image, mft_address + first_offset)
        second_offset = first_offset + first_attribute.get("length")
        second_attribute = self.get_mft_attribute(image, mft_address + second_offset)
        self.mft_attributes = [first_attribute, second_attribute]

    def get_mft_attribute(self, image, offset):
        image.seek(offset)
        attribute_type = self.get_attribute_type(int.from_bytes(image.read(4), "little"))
        attribute_length = int.from_bytes(image.read(4), "little")
        return {"type": attribute_type, "length": attribute_length}

    @staticmethod
    def get_attribute_type(attribute_type):
        if attribute_type == 16:
            return "$STANDARD_INFORMATION"
        elif attribute_type == 32:
            return "$ATTRIBUTE_LIST"
        elif attribute_type == 48:
```

```

        return "$FILE_NAME"
    elif attribute_type == 64:
        return "$OBJECT_ID"
    elif attribute_type == 80:
        return "$SECURITY_DESCRIPTOR"
    elif attribute_type == 96:
        return "$VOLUME_NAME"
    elif attribute_type == 122:
        return "$VOLUME_INFORMATION"
    elif attribute_type == 128:
        return "$DATA"
    elif attribute_type == 144:
        return "$INDEX_ROOT"
    elif attribute_type == 160:
        return "$INDEX_ALLOCATION"
    elif attribute_type == 176:
        return "$BITMAP"
    elif attribute_type == 192:
        return "$REPARSE_POINT"
    elif attribute_type == 256:
        return "$LOGGED_UTILITY_STREAM"
    else:
        return "NOT-DEFINED"

def print_ntfs_info(self):
    print("\n[Volume Information]")
    print(f"Bytes per sector = {self.bytes_per_sector}")
    print(f"Sectors per cluster = {self.sectors_per_cluster}")
    print(f"MFT Sector Address = {self.mft_sector}")
    for i, attribute in enumerate(self.mft_attributes):
        print(f"\n[Attribute {i + 1}]")
        print(f"Type = {attribute.get('type')}")
        print(f"Length (bytes) = {attribute.get('length')}")

```