FAT32 File_System
UnAllocated_Area Analysis

Track	Digital Forensic
Category	Tech_02
Name	이 문 원

목 차

1. 개발 목적 및 목표

1.1 개발 목적

- FAT32 File System의 Unallocated Area를 분석하기 위함이다.
- FAT32 File System의 Unallocated Area에서 이전에 존재했던 파일들의 흔적을 찾는다.
- Unallocated Area에 파일들의 흔적이 존재할 경우, 해당 파일들의 형식을 확인한다.
- 파일들의 형식은 파일 Signature를 이용하여 탐색한다.
- zip 형식의 파일의 경우 docx, pptx, xlsx 파일 형식과 동일하다. 이 때, 더 명확히 구분이 가능하도록 하며, 단순 압축 파일의 경우, 압축된 파일들 중 첫번째 파일 명을 함께 출력한다.

1.2 개발 조건 및 목표.

- 드라이브명 / 볼륨명을 입력 받아 Unallocated Area 탐색
- Unallocated Area에서 파일의 형식 확인이 가능할 경우, 탐색 된 Cluster번호와 파일 형식을 매칭하여 출력한다.
- zip, pdf, jpeg, png 파일 탐색 지원, 파일 형식 탐색의 정확도
- zip 파일 형식에서 docx, pptx, xlsx 등을 더 명확하게 구분
- zip 파일 형식에서 docx, pptx, xlsx가 아닌 경우, 클러스터 사이즈에서 압축된 파일의 첫번째 파일명 함께 출력

2. 개발 진행 시 특이점

2.1 Win32API Module 사용

- 사용자가 입력한 드라이브 볼륨 명에 해당하는 실 드라이브의 존재 확인을 위한 사용한다.
- 드라이브가 존재 할 시, 드라이브의 Format 형식을 확인할 때 사용한다.

2.2 unpack 함수 사용

- Drive에서 읽은 binary hex 값은 각각의 offset에 맞게 Little Endian 형식으로 쓰여 있다.
- 본 함수를 사용하여 Little Endian을 순서대로 읽어 정수형태로 변환한다.

2.3 global 전역 변수 사용

- 특히 VBR, FSINFO에서 parsing 한 offset 값들은 드라이브를 구성하는 기본 정보들이며, 해당 정보들은 드라이브를 읽는 동안 변하지 않는 고정 값을 갖는다.
- 해당 값들은 여러 함수를 호출하면서 전역으로 필요하기 때문에 전역변수로 설정한다.

3. 개발 Script_1 (FAT32_UnAllocArea_analysis.py)

➢ 드라이브 볼륨명의 데이터에 접근하고, 볼륨의 기본 정보와 볼륨의 Unallocated Cluster를 찾은 후, 해당 Cluster의 일정 데이터를 읽고 비교하여, Unallocated Cluster에 파일의 흔적이 있을 경우, 해당 파일의 Format을 확인하는 과정을 아래 서술하였다.

3.1 Main function - Initiate

```
if __name__ == '__main__' :
    drive_name = input("\n======> Write the Logical Drive Name : ")
    if Drive_exist_check(drive_name) == 1 :
        global whole_data
        whole_data = open("\\\\.\\" + drive_name + ":", 'rb')
        print ("\n\t[+] FAT32 VBR Area")
        vbrAREA()
        print ("\n\t[+] FSINFO Area")
        FSINFO_Area()
        print ("\n\t[+] FAT Area 1")
        FATArea()
        print ("\n\t[+] UnAllocated Cluster")
        findUnAllocate()
        whole_data.close()

    else :
        print ("\nProgram is Over")
```

➢ FAT32 File System을 가진 드라이브 볼륨 명을 간단히 입력하도록 input 함수로 구현한다. 예시 입력 : D, E, F, Z ...

➢ 입력한 드라이브 볼륨 명에 해당하는 File System의 여부를 확인함과 동시에 존재 할 경우, File System의 Format 형태가 FAT32인지 확인한다.

➢ 입력한 드라이브 볼륨 명을 PC에서 인식 할 수 있도록 '₩₩.₩₩[drive_name]:'이 되도록 양식을 맞추고, 해당 드라이브 볼륨의 내용을 Binary로 읽어 온다.

➢ 논리 드라이브 명으로 접근하였기 때문에 처음부터 읽는 영역은 MBR (Master Boot Record)가 아닌 VBR (Volume Boot Record)가 된다. 그러므로, VBR 영역부터 FSINFO, FAT 영역 순으로 차례차례 접근하도록 한다.

➢ vbrAREA() : 최초 0x200 Bytes Read = VBR 영역

➢ FSINFO_Area() : VBR 영역에서 확인한 섹터 위치로 이동하여 0x200 Bytes Read = FSINFO

➢ FATArea() : VBR 영역에서 확인한 예약영역의 섹터 수를 바이트로 변환하여 해당 크기 만큼 VBR 영역으로부터 이동한다. 해당 영역은 FAT #1의 영역이다.

➢ findUnAllocate() : FAT #1 영역에서 비 할당 클러스터를 탐색한다.

➢ 예외 처리로 입력한 드라이브 볼륨 명이 존재하지 않을 경우 해당 프로그램을 종료한다.

3.2 Definition Function Drive_exist_check(drive_name)

```
def Drive_exist_check(drive_name) :
    try:
        drive_path = win32api.GetVolumeInformation(drive_name+":\\\\")
        #print(drive_path)
        if drive_check(drive_path[4]) == 1 :
            return 1
        else :
            print ("\nThis Drive FileSystem is not FAT32")
    except :
        print ("Can not find!")
```

- 사용자가 탐색하고자 하는 드라이브 볼륨 명을 입력하였을 경우, 존재 여부를 확인한다
- Win32api에서 제공하는 module을 python에 import 후 입력한 드라이브 볼륨 명이 존재할 경우 해당 드라이브의 정보를 GetVolumeInformation 함수로 호출한다.
- 드라이브 존재 시, list에서 4번째 요소에 해당 드라이브의 Format 형식.을 확인할 수 있다.
- 실제로 드라이브 볼륨 명에 알맞은 드라이브가 있을 경우, drive_check 함수를 호출하여, 앞서 확인한 Format 형식이 FAT32임을 검증한다.
- 드라이브 미 존재 : Can't not find / FAT32가 아닐 시 : This Drive Filesystem is not FAT32

3.3 Definition Function vbrAREA

```
def vbrAREA() :
    global bps, spc, reserved_sector, fatSize32
    vbr = whole_data.read(0x200)
    bps = unpack('<H',vbr[0x0B:0x0D])[0] # Bytes per Sector
    spc = vbr[0x0D] #Sectors per Cluster
    reserved_sector = unpack('<H',vbr[0x0E:0x10])[0] # Result Value's Unit : Sector
    number_of_FatTable = vbr[0x10]
    media_Type = hex(vbr[0x15])
    hidden_Sector = unpack('<L', vbr[0x1C:0x20])[0]
    total_Sector = unpack('<L', vbr[0x20:0x24])[0]
    fatSize32 = unpack('L', vbr[0x24:0x28])[0] # Result Value's Unit : Sector
    rootDir_Cluset_offset = unpack('<L', vbr[0x2C:0x30])[0]
    FSINFO_sector_loc = unpack('<H', vbr[0x30:0x32])[0]
    backup_bootsector_offset = unpack('<H', vbr[0x32:0x34])[0]
    volume_name = unpack('11s', vbr[0x47:0x52])[0]
    FStype = unpack('8s', vbr[0x52:0x5A])[0]
    signature = hex(unpack('>H', vbr[0x1FE:0x200])[0])

    if media_Type == '0xf8' :
        media = 'Disk'
    elif media_Type == '0xf0' :
        media = 'Floppy Disk'
    else :
        print ("I don't know, what is this")

    print ("\t\t[-] Bytes per Sector : %d" % bps)
    print ("\t\t[-] Sectors per Cluster : %d" % spc)
    print ("\t\t[-] Reserved Sector Count : %d" % reserved_sector)
    print ("\t\t\t\t - **** Next Reserved Sector Count is FAT AREA #1 **** ")
    print ("\t\t\t  - **** We Need to Analysis FAT Area #1 for find Allocated Area ****")
    print ("\t\t[-] Media Type : %s (%s)" % (media_Type, media))
    print ("\t\t[-] Hidden Sector Count : %d" % hidden_Sector)
    print ("\t\t[-] FAT32 Size FAT #(1,2) Area Size  : %d sectors (%d Bytes)" % (fatSize32, sector2Bytes(fatSize32)))
    print ("\t\t[-] Root Directory Cluster Offset : %d" % rootDir_Cluset_offset)
    print ("\t\t[-] FSINFO Sector Located : %d%s Sector" % (FSINFO_sector_loc, order_string_set(FSINFO_sector_loc)))
    print ("\t\t[-] backup_bootsector_offset : %d%s Sector" % (backup_bootsector_offset, order_string_set(backup_bootsector_offset)))
    print ("\t\t[-] volume_name : %s" % volume_name)
    print ("\t\t[-] File System Type : %s" % FStype)
    print ("\t\t[-] End signature : %s" % signature)
```

- FAT32 Format의 Volume Boot Record 영역을 탐색하며, 본 드라이브의 기본 정보 획득이 가능하다..

3.3.1 VBR Offset Set

```
global bps, spc, reserved_sector, fatSize32
vbr = whole_data.read(0x200)
bps = unpack('<H',vbr[0x0B:0x0D])[0] # Bytes per Sector
spc = vbr[0x0D] #Sectors per Cluster
reserved_sector = unpack('<H',vbr[0x0E:0x10])[0] # Result Value's Unit : Sector
number_of_FatTable = vbr[0x10]
media_Type = hex(vbr[0x15])
hidden_Sector = unpack('<L', vbr[0x1C:0x20])[0]
total_Sector = unpack('<L', vbr[0x20:0x24])[0]
fatSize32 = unpack('L', vbr[0x24:0x28])[0] # Result Value's Unit : Sector
rootDir_Cluset_offset = unpack('<L', vbr[0x2C:0x30])[0]
FSINFO_sector_loc = unpack('<H', vbr[0x30:0x32])[0]
backup_bootsector_offset = unpack('<H', vbr[0x32:0x34])[0]
volume_name = unpack('11s', vbr[0x47:0x52])[0]
FStype = unpack('8s', vbr[0x52:0x5A])[0]
signature = hex(unpack('>H', vbr[0x1FE:0x200])[0])

if media_Type == '0xf8' :
    media = 'Disk'
elif media_Type == '0xf0' :
    media = 'Floppy Disk'
else :
    print ("I don't know, what is this")
```

- VBR에서 확인 할 수 있는 정보를 각각 Offset에 맞게 parsing 후 각 변수에 저장한다.
- 계속해서 사용될 값을 포함한 bps (bytes per sector), spc (sectors per cluster), reserved_sector, fatSize32는 전역변수로 설정한다.
- VBR에서 확인하는 media type의 경우 저장소가 Disk / Floppy Disk 형식을 확인한다.

3.3.2 VBR Offset Set

- 아래 그림과 같이 각각의 변수를 알맞은 설명에 맞추어 출력한다

```
print ("\t\t[-] Bytes per Sector : %d" % bps)
print ("\t\t[-] Sectors per Cluster : %d" % spc)
print ("\t\t[-] Reserved Sector Count : %d" % reserved_sector)
print ("\t\t\t\t - **** Next Reserved Sector Count is FAT AREA #1 **** ")
print ("\t\t\t  - **** We Need to Analysis FAT Area #1 for find Allocated Area ****")
print ("\t\t[-] Media Type : %s (%s)" % (media_Type, media))
print ("\t\t[-] Hidden Sector Count : %d" % hidden_Sector)
print ("\t\t[-] FAT32 Size FAT #(1,2) Area Size  : %d sectors (%d Bytes)" % (fatSize32, sector2Bytes(fatSize32)))
print ("\t\t[-] Root Directory Cluster Offset : %d" % rootDir_Cluset_offset)
print ("\t\t[-] FSINFO Sector Located : %d%s Sector" % (FSINFO_sector_loc, order_string_set(FSINFO_sector_loc)))
print ("\t\t[-] backup_bootsector_offset : %d%s Sector" % (backup_bootsector_offset, order_string_set(backup_bootsector_offset)))
print ("\t\t[-] volume_name : %s" % volume_name)
print ("\t\t[-] File System Type : %s" % FStype)
print ("\t\t[-] End signature : %s" % signature)
```

3.4 Definition Function FSINFO_Area

```
def FSINFO_Area() :
    global FS_Free_Cluster_Count
    FS_data = whole_data.read(0x200)
    FS_Lead_signature_string = unpack('4s', FS_data[0x00:0x04])[0]
    FS_Lead_signature = unpack('<L', FS_data[0x00:0x04])[0]
    FS_struct_signature_string = unpack('4s', FS_data[0x1E4:0x1E8])[0]
    FS_struct_signature = unpack('<L', FS_data[0x1E4:0x1E8])[0]
    FS_Free_Cluster_Count = unpack('<L', FS_data[0x1E8:0x1EC])[0]
    FS_Next__Free_Cluster_loc = unpack('<L', FS_data[0x1EC:0x1F0])[0]
    FS_Trail_Signature = hex(unpack('>H', FS_data[0x1FE:0x200])[0])
    test = hex(unpack('<H', FS_data[510:512])[0])

    if (hex(FS_Lead_signature) != "0x41615252") and (hex(FS_struct_signature) != "0X61417272") :
        print ('Is not FSINFO Area')
        return 0
    else :
        pass

    print ("\t\t[-] FSINFO Signature_1 : %s(%s)" % (FS_Lead_signature_string, hex(FS_Lead_signature)))
    print ("\t\t[-] FSINFO Signature_2 : %s(%s)" % (FS_struct_signature_string, hex(FS_struct_signature)))
    print ("\t\t[-] FS Free Cluster Count : %d EA" % FS_Free_Cluster_Count)
    print ("\t\t[-] FS Next Free Cluset Location : %d%s Cluster" % (FS_Next__Free_Cluster_loc,
    order_string_set(FS_Next__Free_Cluster_loc)))
    print ("\t\t[-] FS Trail Signature : %s" % FS_Trail_Signature)
```

- VBR 영역에서 FSINFO 영역이 존재하는 Sector 위치 확인 후 해당 위치로 이동한다.
- FSINFO의 경우 FAT32의 예약된 영역의 1번째 Sector이다.
- FSINFO는 첫 4바이트에 Lead Signature와 마지막 Tail Signature를 가지고 있기 때문에 각각 Offset의 값을 비교하여 검증한다.
- FSINFO에는 현재 볼륨에 비어 있는 Cluster의 수와 빈 Cluster들 중 가장 앞 번호의 정보를 포함하고 있다.

3.5 Definition Function FATArea

```
def FATArea() :
    global dataCluster_Count, Non_Data_Cluster, Data_Cluster
    Fat_1_start_loc = sector2Bytes(reserved_sector) # nth Byte of whole disk data Bytes
    sizeFAT = sector2Bytes(fatSize32)
    FA_Custers_Count = int(sizeFAT / 4) # 1 Cluster is expressed by 4Bytes (FAT32 : 4Bytes)
    dataCluster_Count = FA_Custers_Count - 2 # 1st Cluster : Media, 2nd : Partition Status
    whole_data.seek(Fat_1_start_loc)
    Non_Data_Cluster = whole_data.read(0x08)
    Data_Cluster = whole_data.read(sizeFAT-0x08)

    Media_Type_FA1 = hex(unpack('<L', Non_Data_Cluster[0x00:0x04])[0])
    Partition_Stauts = hex(unpack('<L', Non_Data_Cluster[0x04:0x08])[0])

    print ("\t\t[-]FAT #1 Offset : %s" % hex(Fat_1_start_loc))
    print ("\t\t\t[-] Fixed Allcoated Area FAT32")
    print ("\t\t\t\t[-] Media Type : %s" % Media_Type_FA1)
    print ("\t\t\t\t[-] Partition Stauts : %s" % Partition_Stauts)
```

- VBR 영역과 Reserved Area(FSINFO 포함) 이 후의 FAT 영역의 정보를 표시힌다.
- 최초 0번 Offset으로부터 'VBR까지의 데이터 크기 + VBR 영역 + 예약영역'을 한 데이터의 크기 다음에 위치하는 FAT 영역이나, 최초 논리 디스크로 접근하였기 때문에, 0번 Offset은 VBR의 첫 데이터를 의미하므로, 최초 0번으로부터 예약영역의 크기만큼 이동 하면 FAT32 #1의 위치로 이동한다.

- FAT 영역은 #1, #2로 구성되어 있으며, #2는 #1의 내용이 복사된 형태이다.
- FAT 영역에서 현재 볼륨에 할당된 클러스터와 비할당된 클러스터의 정보를 가지고 있다.
- 4Bytes의 단위로 Cluster 0, Cluster 1, ... Cluster n으로 표시하며 사실상 n은 FAT 영역의 크기를 4로 나눈 값으로 생각할 수 있다.
- 1,2 번째 Cluster (Cluster 0, Cluster 1)는 고정적으로 사용되는 영역이다.
- 전체 FAT 영역 중 최초 2개의 Cluster의 크기인 8 Bytes를 제외한 크기를 비할당영역을 탐색할 크기로 지정한다. (변수 명 : Data_Cluster)

3.6 Definition Function findUnAllocated

```
def findUnAllocate() :
    global buf
    buf = 0
    while (buf < FS_Free_Cluster_Count) :
        temp = unpack('4s', Data_Cluster[(buf*4):((buf*4)+4)])[0]
        if temp == b'\x00\x00\x00\x00' :
            into_Cluster(buf)
            buf += 1
        else :
            buf += 1
```

- 비할당영역을 탐색하기 위해 지정한 영역을 4 Bytes 단위로 탐색한다.
- 앞서 구한 n개의 Cluster에서 2개를 제외한 만큼 탐색을 수행하면 된다고 생각 했으나, 뒤쪽 Cluster가 모두 비할당인 것은 아니며, FSINFO에서 확인한 할당되지 않은 Cluster의 개수가 명시 되어 있으므로, 해당 개수만큼 3번재 Cluster (Cluster 2)부터 탐색을 수행한다.
- Unallocated Cluster의 경우 4 바이트 모두 '₩x00'으로 채워져 있다. 모두 0으로 채워진 4 Bytes의 번호를 확인 후 Cluster의 데이터 탐색을 위해 이동한다. (Cluster 7의 경우 Cluster 7의 첫번째 offset으로 이동)
- Unallocated Cluster의 Offset으로 이동 시, into_Cluster 함수를 호출한다.

3.7 Definition Function into_Cluster(cluster_no)

```
def into_Cluster(cluster_no) :
    global file_sig_range, unAllo_cluster_offset
    unAllo_cluster_offset = sector2Bytes((reserved_sector + fatSize32 * 2) + cluster2Sector(buf))
    whole_data.seek(unAllo_cluster_offset)
    file_sig_range = whole_data.read(0x20) # hwp old version's signature is 17bytes
    file_sig_b2 = hex(unpack('<H', file_sig_range[0x00:0x02])[0])
    file_sig_b3 = hex(file_sig_range[0x02]) + file_sig_b2[2:]
    file_sig_b4 = hex(unpack('<L', file_sig_range[0x00:0x04])[0])
    file_sig_b8 = hex(unpack('<Q', file_sig_range[0x00:0x08])[0])
    file_sig_b6 = "0x" + file_sig_b8[6:]
    file_sig_b16_1 = hex(unpack('>QQB', file_sig_range[0x00:0x11])[0])
    file_sig_b16_2 = hex(unpack('>QQB', file_sig_range[0x00:0x11])[1])
    file_sig_b16_3 = hex(unpack('>QQB', file_sig_range[0x00:0x11])[2])
    file_sig_b16 = file_sig_b16_1 + file_sig_b16_2[2:] + file_sig_b16_3[2:]

    if file_sig_b2 == "0x5a4d" : # MZ
        result = check_signature_exe(file_sig_b3, file_sig_b8)
        if result == 1 :
            print(UnAlloc_print() + "MZ Executable File")
        else :
            print (UnAlloc_print() + result)
    elif file_sig_b2 == "0x4d42" :
        print (UnAlloc_print() + ".bmp (Windows Bitmap Image)")
    elif signature_print(file_sig_b3) != 0 :
        print (UnAlloc_print() + signature_print(file_sig_b3))
    elif signature_print(file_sig_b4) != 0 :
        if file_sig_b4 == "0x4034b50" : # PK
            #print (ext_pk_analysis())
            if ext_pk_analysis() == 'zip' :
                print (UnAlloc_print() + ext_pk_analysis() + " {" +filename_inzip() + "}")
            else :
                print (UnAlloc_print() + ext_pk_analysis())
        elif signature_print(file_sig_b6) != 0 :
            print (UnAlloc_print() + signature_print(file_sig_b6))
    elif signature_print(file_sig_b6) != 0 :
        print (UnAlloc_print() + signature_print(file_sig_b6))
    elif (signature_print(file_sig_b8)) != 0 :
        print (UnAlloc_print() + signature_print(file_sig_b8))
    elif file_sig_b16 == "0x48575020446f63756d656e742046696c65" : # hwp old version signature (97 ~ 3.0)
        print (UnAlloc_print() + "hwp (97 ~ 3.0 old version)")
```

- 파일을 드라이브에 쓸 경우, 파일의 크기를 Cluster 단위로 할당하여 사용한다. 또한 현재 사용되는 Cluster 이후에 있는 Unallocated Cluster 부터 파일을 쓴다.
- VBR 영역에서 Sectors per Cluster = 8, Bytes per Sector = 512라고 하면, 1 Cluster = 8 Sectors = 4096(8*512)Bytes가 된다. 총 파일의 크기를 4096 Bytes로 나누고, 나온 몫이 정수라면 해당 몫의 개수만큼, 정수가 아니면 정수 +1개 만큼 Cluster를 할당한다.
- 어떤 Cluster의 번호부터 할당을 하더라도 할당된 Cluster 들 중 1번째 Cluster의 0번째 Offset부터 일정 크기만큼 쓰는 파일의 Signature가 존재할 것이다.
- 디스크 상에서 파일을 지우면 할당된 Cluster가 할당 해제 되면서 FAT 영역의 Cluster 정보는 0으로 변경되지만 지운 파일에 할당되었던 Cluster를 다른 파일이 덮어 쓰지 않는 이상, 파일의 데이터는 온전히 남아 있다.
- 4 Bytes 모두 0인 Unallocated Cluster는 단순히 아직 한번도 사용되지 않은 Cluster인지, 파일이 있었던 Cluster인지는 확인이 불가능하기 때문에, 각 Cluster의 위치로 이동하여 데이터를 읽고, 여러 파일들의 Signature들을 비교하여 일치할 경우, 데이터가 어떤 파일의 형태로 있었는지 확인이 가능하다.

3.7.1 Set the Variable for Verification File Signature

```
def into_Cluster(cluster_no) :
    global file_sig_range, unAllo_cluster_offset
    unAllo_cluster_offset = sector2Bytes((reserved_sector + fatSize32 * 2) + cluster2Sector(buf))
    whole_data.seek(unAllo_cluster_offset)
    file_sig_range = whole_data.read(0x20) # hwp old version's signature is 17bytes
    file_sig_b2 = hex(unpack('<H', file_sig_range[0x00:0x02])[0])
    file_sig_b3 = hex(file_sig_range[0x02]) + file_sig_b2[2:]
    file_sig_b4 = hex(unpack('<L', file_sig_range[0x00:0x04])[0])
    file_sig_b8 = hex(unpack('<Q', file_sig_range[0x00:0x08])[0])
    file_sig_b6 = "0x" + file_sig_b8[6:]
    file_sig_b17_1 = hex(unpack('>QQB', file_sig_range[0x00:0x11])[0])
    file_sig_b17_2 = hex(unpack('>QQB', file_sig_range[0x00:0x11])[1])
    file_sig_b17_3 = hex(unpack('>QQB', file_sig_range[0x00:0x11])[2])
    file_sig_b17 = file_sig_b17_1 + file_sig_b17_2[2:] + file_sig_b17_3[2:]
```

- 파일들의 Signature들은 2 Bytes 부터 많게는 17 Bytes까지 다양하기 때문에, 여러 조건을 충족 시키고자 2, 3, 4, 6, 8, 17 Bytes를 검증할 수 있도록 변수를 설정한다. 각 변수들은 Cluster의 데이터를 2, 3, 4, 6, 8 Bytes씩 Little Endian 방식 그대로 Hex형태로 저장하고, 17 Bytes는 앞에서 순차적으로 읽어 저장한다.

3.7.2 Algorithm for Verification File Signature

```
if file_sig_b2 == "0x5a4d" : # MZ
    result = check_signature_exe(file_sig_b3, file_sig_b8)
    if result == 1 :
        print(UnAlloc_print() + "MZ Executable File")
    else :
        print (UnAlloc_print() + result)
elif file_sig_b2 == "0x4d42" :
    print (UnAlloc_print() + ".bmp (Windows Bitmap Image)")
elif signature_print(file_sig_b3) != 0 :
    print (UnAlloc_print() + signature_print(file_sig_b3))
elif signature_print(file_sig_b4) != 0 :
    if file_sig_b4 == "0x4034b50" : # PK
        #print (ext_pk_analysis())
        if ext_pk_analysis() == 'zip' :
            print (UnAlloc_print() + ext_pk_analysis() + " {" +filename_inzip() + "}")
        else :
            print (UnAlloc_print() + ext_pk_analysis())
    elif signature_print(file_sig_b6) != 0 :
        print (UnAlloc_print() + signature_print(file_sig_b6))
elif signature_print(file_sig_b6) != 0 :
    print (UnAlloc_print() + signature_print(file_sig_b6))
elif (signature_print(file_sig_b8)) != 0 :
    print (UnAlloc_print() + signature_print(file_sig_b8))
elif file_sig_b17 == "0x48575020446f63756d656e742046696c65" : # hwp old version signature (97 ~ 3.0)
    print (UnAlloc_print() + "hwp (97 ~ 3.0 old version)")
```

- 변수들의 크기만큼 읽어서 저장한 Cluster의 데이터들을 하나씩 비교하여, 일치하는 경우 해당 Cluster를 사용했던 파일 데이터의 파일 Format을 출력한다.

- 특정 파일 Signature는 파일의 확장자가 달라도 같을 경우가 존재한다. 이는 표현하는 방식만 다를 뿐 같은 형태의 파일임을 의미한다. 검증한 File Signature가 충분히 여러 파일 Format을 나타낼 경우 더욱 세분화(본 Script에서 실행 파일의 경우 check_sigature_exe 함수를, 압축 파일의 경우 ext_pk_analysis 함수를 사용)하여 정밀검증을 하도록 하였다.

- Cluster에 존재하는 데이터들을 2, 3, 4, 6, 8, 17 Bytes 순으로 조건에 충족하지않을 경우 검증하도록 순차적 접근 방식을 채택하였다.

4. 개발 Script_2 (FAT32_UnAllocArea_analysis.py)

➢ Unallocated Cluster의 Data Offset에 접근 후, 특정 데이터를 읽어 저장 후, 많은 File Signature들과 비교하여 해당 Cluster에 어떤 Format의 .파일이 있었는지 검증하는 법을 서술하였다.

4.1 실행파일 (MZ or 0x4D5A(Big Endian))의 세부 검증

```
if file_sig_b2 == "0x5a4d" : # MZ
    result = check_signature_exe(file_sig_b3, file_sig_b8)
    if result == 1 :
        print(UnAlloc_print() + "MZ Executable File")
    else :
        print (UnAlloc_print() + result)
elif file_sig_b2 == "0x4d42" :
    print (UnAlloc_print() + ".bmp (Windows Bitmap Image)")
```

```
def check_signature_exe(bytes3, bytes8) :
    if bytes3 == '0x905a4d' :
        return "exe (Microsoft Executable)"
    elif bytes8 == '0x300905a4d' :
        return ".acm (Executable) or .dll (Dynamic Link Library)"
    else :
        return 1
```

➢ '0x4D5A'의 경우 일반적으로 실행파일의 File Signature이다.

➢ '0x4D5A'로 시작하는 실행파일도 여러 종류가 존재하기 때문에, 예외 조건을 추가하여 검증을 실시 한다. – check_signature_exe 함수 호출

➢ 첫 2 Bytes가 '0x4D5A'이면 3 Byte를 읽은 값과 8 Byte를 한번 더 검증한다.
3 Bytes가 '0x4D5A90'이면 확장자가 exe인 Microsoft 규격의 실행파일, 8 Bytes의 값이 '0x4D5A900003'의 경우 확장자가 acm 또는 dll 파일이다.

➢ 2 Bytes가 '0x4D5A'가 아니면서 '0x424D'인 경우 Window Bitmap Image임을 확인할 수 있다.

4.2 압축파일 (PK or 0x504B0304(Big Endian))의 세부 검증

```
elif signature_print(file_sig_b4) != 0 :
    if file_sig_b4 == "0x4034b50" : # PK
        #print (ext_pk_analysis())
        if ext_pk_analysis() == 'zip' :
            print (UnAlloc_print() + ext_pk_analysis() + " {" +filename_inzip() + "}")
        else :
            print (UnAlloc_print() + ext_pk_analysis())
    elif signature_print(file_sig_b6) != 0 :
        print (UnAlloc_print() + signature_print(file_sig_b6))
```

➢ 4 Bytes File Signature를 검색하였을 경우, '0x504B0304'는 zip 압축 형태의 파일을 의미한다.

➢ Microsoft Office에서 제공하는 docx, xlsx, pptx의 경우 파일 Signature의 4 Bytes는 zip file Signature와 동일하기 때문에 단순 4 Bytes 비교로는 정확한 파일 Format 확인이 힘들다.

```
def ext_pk_analysis () :
    pk_classfi = hex(unpack('>L', file_sig_range[0x04:0x08])[0])
    if pk_classfi == "0x14000600" : # zip / docx, pptx, xlsx
        whole_data.seek(unAllo_cluster_offset)
        file_data = whole_data.read(0xA00) # https://kldp.org/node/141380
        #print (file_data)
        if 'word/document' in str(file_data) :
            return "zip [docx] (MS Word 2007+)"
        elif 'xl/worksheets/' in str(file_data) :
            return "zip [xlsx] (MS Excel 2007+)"
        elif 'ppt/slides' in str(file_data) :
            return "zip [pptx] (MS PowerPoint 2007+)"
        else :
            return "zip"
    else :
        whole_data.seek(unAllo_cluster_offset)
        file_data = whole_data.read(0xA00)
        #print (file_data)
        if 'xl/worksheets/' in str(file_data) :
            return "zip [xlsx] (MS Excel 2007+)"
        else :
            return "zip"
```

- 4 Bytes의 Signature를 검증 하였을 경우 '0x504B0304'에 해당하면, 압축 형태의 파일들로 인식 후 세부 분석을 위하여 exe_pk_analysis 함수를 호출한다.
- 최초 검증 Offset Bytes 이후 추가 4 Bytes를 검증하여 실제 압축 한 zip 파일과 Microsoft에서 제공하는 파일형태로 분류가 가능하다.
- '0x14000600'의 파일 데이터가 추가 검증한 4 Bytes에 존재 할 경우에도 Microsoft에서 제공되는 파일의 양식은 모두 동일하다.
  MS Word(docs), PowerPoint(pptx), MS Excel(xlsx)의 File Signature = '0x504B030414000600.
- 검증 단계에서 xlsx 확장자의 파일의 File Signature가 완전히 zip 파일과 같은 경우가 존재하였다. 이에 xlsx의 특징을 선정하고 필터를 통해 zip과의 구별을 하였다.

4.2.1 Microsoft Office File Format 구별

- https://kldp.org/node/141380를 참조.

```
>>>&26" rel="nofollow">http://technet.microsoft.com/en-us/library/cc179224.aspx
>>>&26</a>          string          word/           Microsoft Word 2007+
!:mime application/vnd.openxmlformats-officedocument.wordprocessingml.document
>>>&26          string          ppt/            Microsoft PowerPoint 2007+
!:mime application/vnd.openxmlformats-officedocument.presentationml.presentation
>>>&26          string          xl/             Microsoft Excel 2007+
!:mime application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
>>>&26          default         x               Microsoft OOXML
```

- MS Word의 경우 'word/'라는 문자열, MS PowerPoint의 경우 'ppt/', MS Excel의 경우 'xl/'의 문자열을 포함하고 있다.

```
33 9B 0F 00 00 00 FF FF 03 00 50 4B 03 04 14 00   3›....ÿÿ..PK....
06 00 08 00 00 00 21 00 4B F5 3D EC BD 00 00 00   ......!.Kõ=ì½...
37 01 00 00 20 00 00 00 70 70 74 2F 73 6C 69 64   7... ...ppt/slid
65 73 2F 5F 72 65 6C 73 2F 73 6C 69 64 65 31 2E   es/_rels/slidel.
78 6D 6C 2E 72 65 6C 73 8C CF BD 0A C2 30 10 07   xml.relsŒÏ½.Â0..
```

< MS PowerPoint >

```
E3 D3 24 F0 D5 C7 9C FD 02 00 00 FF FF 03 00 50   ãÓ$ðÕÇœý...ÿÿ..P
4B 03 04 14 00 06 00 08 00 00 00 21 00 1D 84 0A   K..........!..„.
31 07 10 00 00 9D 75 00 00 11 00 00 00 77 6F 72   1.....u.....wor
64 2F 64 6F 63 75 6D 65 6E 74 2E 78 6D 6C EC 1D   d/document.xmlì.
6B 6F D3 58 F6 FB 4A FB 1F AE F2 61 3E 51 62 C7   koÓXöûJû.®òa>QbÇ
```

< MS Word >

```
8B BF 01 50 4B 03 04 14 00 00 00 08 00 74 31 3F   ‹¿.PK........t1?
4C D9 B2 9A 32 9F 01 00 00 15 03 00 00 18 00 00   LÙ²š2Ÿ..........
00 78 6C 2F 77 6F 72 6B 73 68 65 65 74 73 2F 73   .xl/worksheets/s
68 65 65 74 31 2E 78 6D 6C 6D 53 DB 6A DC 30 10   heet1.xmlmSÛjÜ0.
FD 15 A1 0F 88 BC A6 49 C3 62 1B BA 09 A5 85 14   ý.¡.ˆ¼¦IÃb.º.¥….
```

< MS Excel >

- Slide라는 단어는 MS Office Program 중 PowerPoint에서만 사용하고, sheet는 Excel에서만 사용하는 단어, word는 document이기 때문에 앞서 언급한 url 구별에 추가한다

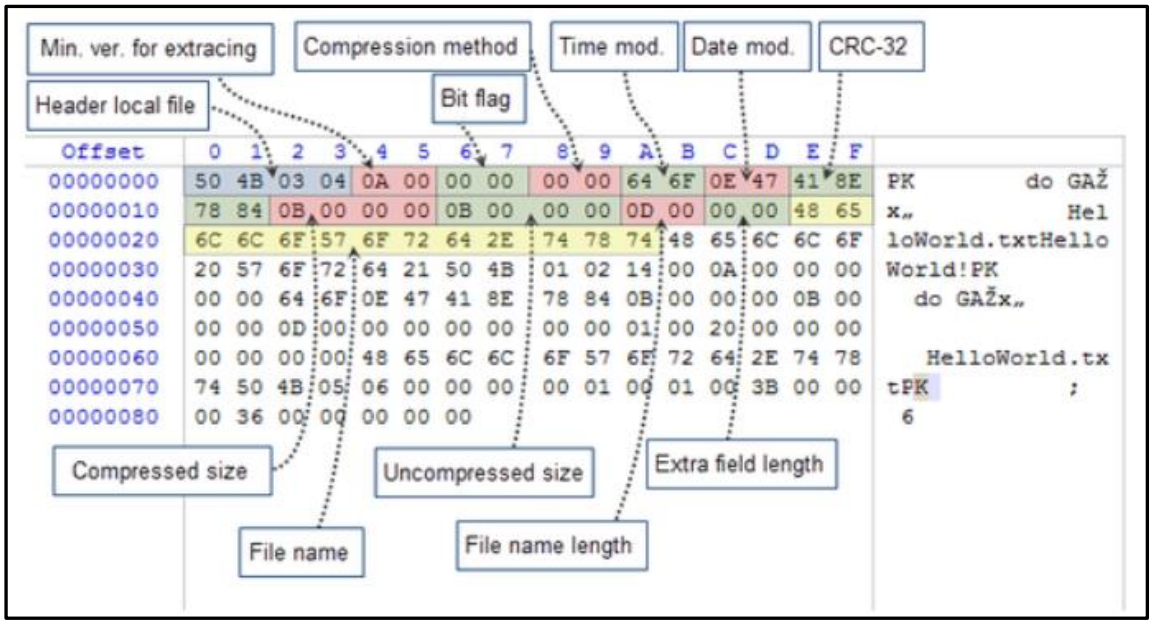
4.3 일반 zip File일 경우 압축된 파일들 중에서 첫번째 파일명 출력

- http://blog.naver.com/PostView.nhn?blogId=koromoon&logNo=220612641115&parentCategoryNo=&categoryNo=&viewDate=&isShowPopularPosts=false&from=postView

```
def filename_inzip () :
    whole_data.seek(unAllo_cluster_offset)
    zipfileHeader = whole_data.read(0x1E)
    #print (zipfileHeader)
    Header_sig = hex(unpack('>L', zipfileHeader[0x00:0x04])[0])
    filename_len = unpack('<H', zipfileHeader[0x1A:0x1C])[0]

    if Header_sig == "0x504b0304" :
        whole_data.seek(unAllo_cluster_offset + 0x1E)
        return ("first file name : " + whole_data.read(filename_len).decode('euc-kr'))
```

- zip File의 처음 30 Bytes는 압축된 파일들 중 첫번째 파일의 파일명 길이가 포함되어 있다.
- 파일명은 30 Bytes 이후 파일명 길이만큼 저장되어 있다. Header이후 파일명 길이 만큼 출력하면 처음 파일명 이름이 출력된다.
- 아래는 위 명시한 url에서 참고한 ZIP File Header이다.



4.4 File Signature Set

```
def signature_print(sig) :
    global sig_b3, sig_b4, sig_b6, sig_b8
    sig_b3 = {"0x685a42" : "bz or bz2 (Bzip Archive)", "0x88b1f" : "gz (GZ Compressed File)",
    "0x2a4949" : "tif or tiff (Little Endian)", "0x2a4d4d" : "tif or tiff (Big Endian)"}

    sig_b4 = {"0x20495641" : "avi", "0x46464952" : "avi", "0x35706733" : "mp4 (MPEG-4 Video File)",
    "0xe1ffd8ff" : "jpg (JPG Graphical File)", "0xe0ffd8ff" : "JPG (JPG Graphical File)","0x3334449" : "mp3 (MP3 Audio)",
    "0x1b3" : "mpg or mpeg (MPEG Movie)", "0x7461646d" : "mov (QuickTime Movie)", "0x766f6f6d" : "mov (QuickTime Movie)",
    "0x3e8000f" : "ppt (PowerPoint Presentation)", "0x2a004949" : "tif or tiff (Little Endian)", "0x2a004d4d" : "tif or tiff (Big Endian)",
    "0xfeffd8ff" : "jpeg (JPG Graphical File)", "0x15a4c41" : "alz (ESTsoft Alzip Archive)", "0x4034b50" :"Archive files"}

    sig_b6 = {"0x613738464947" : "gif (Graphics Interchange Format)", "0x613938464947": "gif (Graphics Interchange Format)",
    "0x70695a6e6957" : "winzip (Winzip Archive)", "0x4554494c4b50" : "zip (PKLITE ZIP Archive)"}

    sig_b8 = {"0x1426a46464952" : "avi (Audio Video interleave File", "0x30322f32312f335b" : "bak (BackUp)",
    "0x6d6d7544204d4552" : "bat (Batch File)", "0x50414dff434353ff" : "bin (Binary File)", "0x6576206c6d783f3c" : "config (xml config file)",
    "0x461c0d3e0002014c" : "exp (Export File)", "0x505954434f44213c" : "htm (HyperText Markup)",
    "0x6576206c6d783f3c" : "msc (Microsoft Magement Console Snap-in Control File", "0x10000000060409" : "xml (MS Excel)",
    "0x332e312d46445025" : "pdf (Adobe Portable Document File)", "0x342e312d46445025" : "pdf (Adobe Portable Document File)",
    "0xa1a0a0d474e5089" : "png (Portable Network Graphic)", "0x4143435300000011" : "pf (Windows Prefetch File)",
    "0x7079746618000000" : "mp4 (MPEG-4 Video File)", "0x707974661c000000" : "mp4 (MPEG-4 Video File)",
    "0x8001404034b50" : "jar (Java Archive)", "0xe11ab1a1e011cfd0" : "hwp"}

    if sig in sig_b3.keys() :
        return sig_b3[sig]
    elif sig in sig_b4.keys() :
        return sig_b4[sig]
    elif sig in sig_b6.keys() :
        return sig_b6[sig]
    elif sig in sig_b8.keys() :
        return sig_b8[sig]
    else :
        return 0
```

- 각 파일들의 signature 값들이 있으며, Unallocated Area에서 데이터를 읽고 파일 Format을 검증할 때 해당 함수 (signature_print)를 호출하여 File Signature Dictionary에서 일치하는 값이 있는지 탐색을 수행한다.

5. 실행 결과

5.1 VBR Area

```
C:₩Users₩L3ad0Xff₩Desktop₩BoB_Stage_3₩Tech₩#02 FAT>python FAT32_UnAllocArea_analysis.py

======> Write the Logical Drive Name : z

        [+] FAT32 VBR Area
                [-] Bytes per Sector : 512
                [-] Sectors per Cluster : 8
                [-] Reserved Sector Count : 8234
                                - **** Next Reserved Sector Count is FAT AREA #1 ****
                        - **** We Need to Analysis FAT Area #1 for find Allocated Area ****
                [-] Media Type : 0xf8 (Disk)
                [-] Hidden Sector Count : 128
                [-] FAT32 Size FAT #(1,2) Area Size  : 4075 sectors (2086400 Bytes)
                [-] Root Directory Cluster Offset : 2
                [-] FSINFO Sector Located : 1st Sector
                [-] backup_bootsector_offset : 6th Sector
                [-] volume_name : b'NO NAME    '
                [-] File System Type : b'FAT32   '
                [-] End signature : 0x55aa
```

5.2 FSINFO Area

```
[+] FSINFO Area
        [-] FSINFO Signature_1 : b'RRaA'(0x41615252)
        [-] FSINFO Signature_2 : b'rrAa'(0x61417272)
        [-] FS Free Cluster Count : 521466 EA
        [-] FS Next Free Cluset Location : 6th Cluster
        [-] FS Trail Signature : 0x55aa
```

5.3 FAT Area (Fixed Cluster Information)

```
[+] FAT Area 1
        [-]FAT #1 Offset : 0x405400
                [-] Fixed Allcoated Area FAT32
                        [-] Media Type : 0xffffff8
                        [-] Partition Stauts : 0xffffffff
```

5.4 Unallocated Cluster Search & Find File Format by File Signature (Option : zip – filename)

```
[+] UnAllocated Cluster
        [-] Cluster 6 - png (Portable Network Graphic)
        [-] Cluster 92 - zip [pptx] (MS PowerPoint 2007+)
        [-] Cluster 107 - zip [docx] (MS Word 2007+)
        [-] Cluster 361 - zip {first file name : syslog-ng.conf}
        [-] Cluster 362 - hwp
        [-] Cluster 364 - hwp (97 ~ 3.0 old version)
        [-] Cluster 395 - exe (Microsoft Executable)
        [-] Cluster 1867 - exe (Microsoft Executable)
        [-] Cluster 5261 - zip [xlsx] (MS Excel 2007+)
        [-] Cluster 5263 - zip {first file name : photo1.jpg}
        [-] Cluster 11611 - mp4 (MPEG-4 Video File)
        [-] Cluster 15597 - .bmp (Windows Bitmap Image)

------*------Finish!!------*-------
```

6. 추가 내용

- Root Directory를 접근하고, 존재하는 모든 data Entry에 대하여, 분류를 수행한 후에, 파일이 위치했던 Cluster의 high, low offset을 읽은 후, Unallocated Area를 거쳐서 검증한 값과 교차검증을 이용하면 좀 더 정확하지 않을까?

  → 단순히 Unallocated Cluster가 한번만 사용되었다면 가능하지만, 여러 번 덮어 쓰였을 경우에는 data Entry에서 중복되는 Cluster 번호들이 많을 것이다. 가장 마지막에 기록된 data Entry에 해당하는 파일 정보일 것이라 생각한다. Data Entry에는 저장되었던 파일의 확장자가 저장되어 있기 때문에, File signature에 맞는 파일 확장자와 교차검증이 가능하다고 판단.

- 한글 문서의 경우, old version이 아님에도 불구하고 old version File Signature가 검색이 된다.

  → 옛 한글 문서와의 호환성 문제로 현재 hwp의 데이터에 삽입 되어 있는 것으로 판단된다. 'Carving 시, 구 버전의 File Signature부터 진행하면 예전 버전의 한글 프로그램에서만 열리지 않을까?'라는 의문점.

7. 참고문헌

- FAT32 File System : http://blog.naver.com/PostView.nhn?blogId=bitnang&logNo=70183899277
- File Signature : http://forensic-proof.com/archives/300
- File Signature 중 HWP old Version : https://sinarn.blog.me/130156996913
- MS Office 문서 구별 방안 : https://kldp.org/node/141380.
- ZIP File Header Analysis : http://blog.naver.com/PostView.nhn?blogId=koromoon&logNo=220612641115&parentCategoryNo=&categoryNo=&viewDate=&isShowPopularPosts=false&from=postView