**그래픽, 클립아트, 그림, 디자인이(가) 표시된 사진

자동 생성된 설명**

**[BMP Recovery] Write-Up**

|  |  |
| --- | --- |
| **작성자** | 심주완 |
| **분석 일자** | 2024.05.10 |
| **작성 일자** | 2024.05.10 |
| **분석 대상** | bmp파일 |
| **문서 버전** | 2 |
| **작성자 E-mail** | [rd002@naver.com](http://rd002@naver.com) |

1. **목차**

[**1.** **문제** 3](#_Toc165902295)

[**2.** **분석 도구** 3](#_Toc165902296)

[**3.** **환경** 3](#_Toc165902297)

[**4.** **Write-Up** 4](#_Toc165902298)

[**5.** **Flag** 5](#_Toc165902299)

[**6.** **별도 첨부** 6](#_Toc165902300)

[**7.** **Reference** 7](#_Toc165902301)

# **문제**

|  |  |
| --- | --- |
| **URL** | <https://dreamhack.io/wargame/challenges/1187> |
| **문제**  **내용** | BMP 파일에서 이런 중요한 값들을 지워버리다니!  빨리 복구해서 플래그를 읽어주세요!  플래그 형식은 DH{...}로, flag.bmp를 올바르게 복구하면 찾을 수 있습니다. |
| **문제**  **파일** |  |
| **문제**  **유형** | multimedia forensics |
| **난이도** | 3 / 5 |

# **분석 도구**

|  |  |  |
| --- | --- | --- |
| **도구명** | **다운로드 링크** | **Version** |
| HxD | <https://hxd.softonic.kr/> | 2.5.0.0 |
| VScode | <https://code.visualstudio.com/> | 1.89 |
|  |  |  |

# **환경**

|  |
| --- |
| **OS** |
| Windows 11 Home |

# **Write-Up**

|  |  |
| --- | --- |
| **파일명** | flag.bmp.broken |
| **용량** | 13976KB |
| **SHA256** | 1CD47A0BC682774F9321EFAA43407F5E76AB4EF8B71C3414601FB81DFC42F56F |
| **Timestamp** | 2023-11-10 16:25:34 |

문제 난이도는 쉽지 않았다. flag.bmp 파일에 chal.py 파일을 적용시켜 파일을 회손한 뒤flag.bmp.broken 파일로 만들어 이 파일을 다시 복원하는 것이 문제의 핵심이었다.

chal.py파일을 먼저 확인해보자.

with open('flag.bmp', 'rb') as f:

    data = bytearray(f.read())

data[:0x1C] = b'\x00' \* 0x1C

data[0x22:0x36] = b'\x00' \* 0x14

with open('flag.bmp.broken', 'wb') as f:

    f.write(data)

0x00 ~ 0x1C 를 00으로 덮었고, 0x22 ~ 0x36을 00으로 덮었다. 이 부분을 복원해야한다.

덮힌 부분을 눈으로 확인하기 위하여 HxD를 활용하여 flag.bmp.broken을 열어보았다.

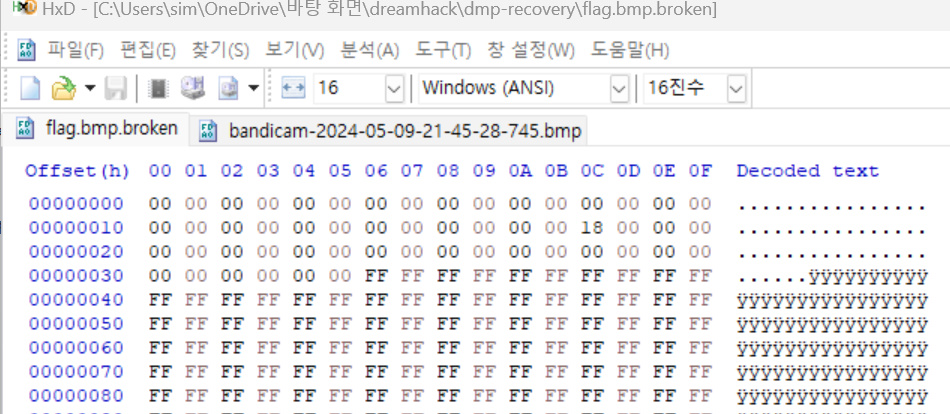


Figure 1 HxD로 열어본 flag.bmp.broken

다음과 같은 부분이 덮혀있을 확인할 수 있었다. 많은 부분이 덮혔지만 모든 부분이 00으로 덮히지 않은 것을 확인할 수 있다. **00으로 덮히는 부분 중 0x1C ~ 0x1D부분은 덮히지 않았다.** 그렇다면 덮힌 부분은 무엇인지, 그중에서 덮히지 않은 부분과의 관계가 무엇인지 알아보자.

앞쪽 부분이 덮힌 것으로 봐서 헤더 부분이 덮혔을 것이라고 예상하였고, 헤더 부분이 맞았다. 찾은 정보는 7. Reference에 넣어두었다. 덮힌 정보는 Header 부분과 InfoHeader 부분이었고, 0x1C ~ 0x21 부분이었다. 이 부분은 BitsPerPixel 이라는 부분과 Compression 이었다. 각 부분을 하나하나 생각해보자.

첫 번째로 BitsPerPixel 부분에서 파일 깂은 2bit4인 것을 확인할 수 있다. BitsPerPixel은 픽셀 하나를 표현하기 위해 필요한 비트 개수를 나타내는데, 값이 다섯 가지가 사용된다.

1: 흑과 백 두 가지 색상만 존재한다.

2: 총 16가지의 색상이 존재한다. 16가지의 색상은 InfoHeader 뒤에 ColorTable 영역에서 정의한다.

4: 총 256가지의 색상이 존재한다.

16: 16비트 RGB로 색상을 나타낸다. RGB565로도 불립니다.

24: 24비트 RGB로 색상을 나타낸다. Red, Green, Blue에 각각 1바이트를 사용하는 일반적인 방식이다.

이 파일에서는 24가 사용되었고, RGB를 고려하여 하나에 픽셀에 총 3바이트가 쓰이는 것을 알 수 있었다. 이 정보를 이용하여 파일의 가로와 세로의 길이를 구할 수 있는데,

이미지의 크기 = 파일의 크기 – 헤더의 크기 이고,

픽셀의 수 = 이미지 파일의 크기 / 3이다.

**픽셀의 수 = 가로 x 세로 이고, 가로, 세로 값은 곧 00으로 덮힌 헤더 값 중 하나이다.**

두 번째로 Compression은 이미지를 압축하는 방법을 나타낸다. 0이기 때문에 무압축 모드이다. 무압축 모드이면 ImageSize(파일의 크기 – 헤더 파일의 크기)가 압축이 들어가지 않기 때문에 그대로 넣으면 덮힌 헤더 파일 중 Imageisze 부분을 복원할 수 있다.

그렇다면 실제 값들을 구해보자.



Figure 2 flag.bmp.broken 속성값

속성 옵션을 통하여 크기를 구하였다. 14,309,622바이트이고, 헤더의 크기는 Header와 InfoHeader값을 더하면 54바이트임을 확인할 수 있다.

그렇다면 이미지 파일의 값은

14,309,622 - 헤더 = 14,309,568

임을 확인할 수 있다.

총 픽셀 사용량의 치수는 픽셀을 하나에 3바이트로 두고 14309568/3을 하면 4,769,856을 구할 수 있다. **그렇다면 높이 x 너비가 4,769,856라는 것인데 더 이상 좋은 방법이 떠오르지 않아 brute-force를 사용하여 될 수 있는 모든 높이와 너비의 bmp 파일을 구하는 코드를 작성하였다.**

def brute\_force():

    for i in range(1,pixels,1):

        if(i\*i>pixels):

            break

        if(pixels%i!=0):

            continue

        recover\_img(i,pixels//i)

        recover\_img(pixels//i,i)

brute\_force()

모든 계산을 하면 파일을 찾을 때 너무 많은 시간이 걸릴 것 같아 결국 소수를 판별하는 일이기 때문에 i\*i>pixels일 경우와 소수가 아닌 경우 연산을 그만두게 하였다.

이제 가로와 세로의 값을 구했으니, 나머지 헤더에 넣을 수 있는 값들을 레퍼런스에 주어진 값들을 비교하며 체크해보자.

시그니처 값은 임의의 bmp파일을 넣어서 쉽게 구할 수 있었다.

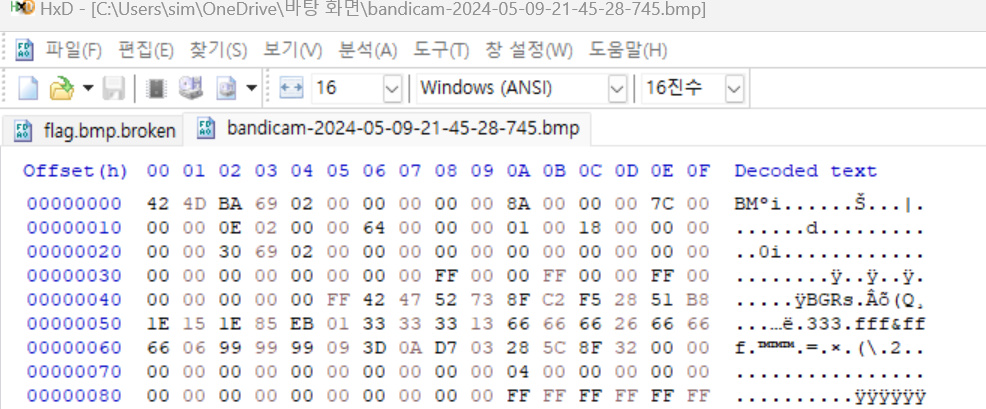


Figure 3 임의의 bmp 파일

정리를 해보면 복원 가능한 데이터는 **시그니처 (0x00) -> 42 4D, FullSize(0x02), DataOffset(0x0A)->0x36, Size(0x0E)->40, Width(0x12), Height(0x16), Planes(0x1A)->1, ImageSize(파일 사이즈-헤더 크기)**가 되겠다.

이를 모두 통합하여 이미지 파일을 만드는 코드를 작성해보자.

with open('flag.bmp.broken','rb') as f:

    data = f.read()

f\_size=len(data)

i\_size=len(data)-0x36

pixels=i\_size//3

def recover\_img(w,h):

    format\_header = """

    42 4D {format\_f\_size} 00 00 00 00 36 00 00 00 28 00

    00 00 {format\_w} {format\_h} 01 00 18 00 00 00

    00 00 {format\_i\_size} 00 00 00 00 00 00 00 00 00 00

    00 00 00 00 00 00

    """

    header=format\_header.format(

        format\_f\_size=f\_size.to\_bytes(4,'little').hex(),

        format\_i\_size=i\_size.to\_bytes(4,'little').hex(),

        format\_w=w.to\_bytes(4,'little').hex(),

        format\_h=h.to\_bytes(4,'little').hex()

    )

    header = bytes.fromhex(header.replace('\n','').replace(' ',''))

    with open(f'recovered\_img/{w},{h}.bmp','wb') as f:

        f.write(header)

        f.write(data[len(header):])

def brute\_force():

    for i in range(1,pixels,1):

        if(i\*i>pixels):

            break

        if(pixels%i!=0):

            continue

        recover\_img(i,pixels//i)

        recover\_img(pixels//i,i)

brute\_force()

recover\_img 함수 부분을 확인해보면 주어진 가로, 세로 값을 활용하고, 앞에서 명시한 넣을 수 있는 값들을 모두 넣은 뒤에 bmp 파일을 생성하여 recovered\_img 폴더에 넣은 것을 확인할 수 있다.

이제 recovered\_img 폴더를 열어보자.

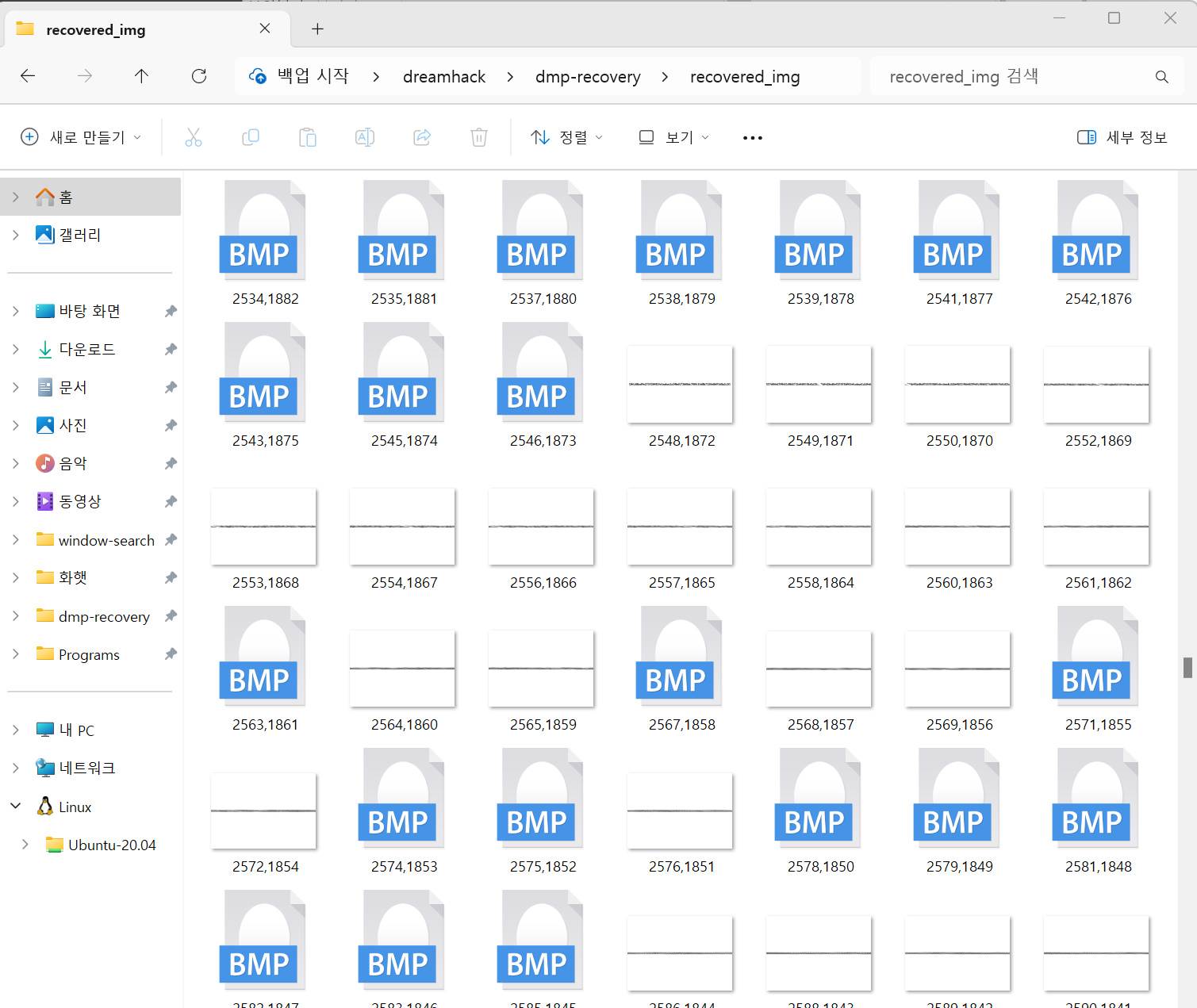


Figure 4 recovered\_img 파일 내부

생성되는 bmp 파일을 줄였음에도 불구하고 상당히 많은 파일이 생성되었다. 찾는데에 시간이 오래 걸렸지만 찾아낼 순 있었다.

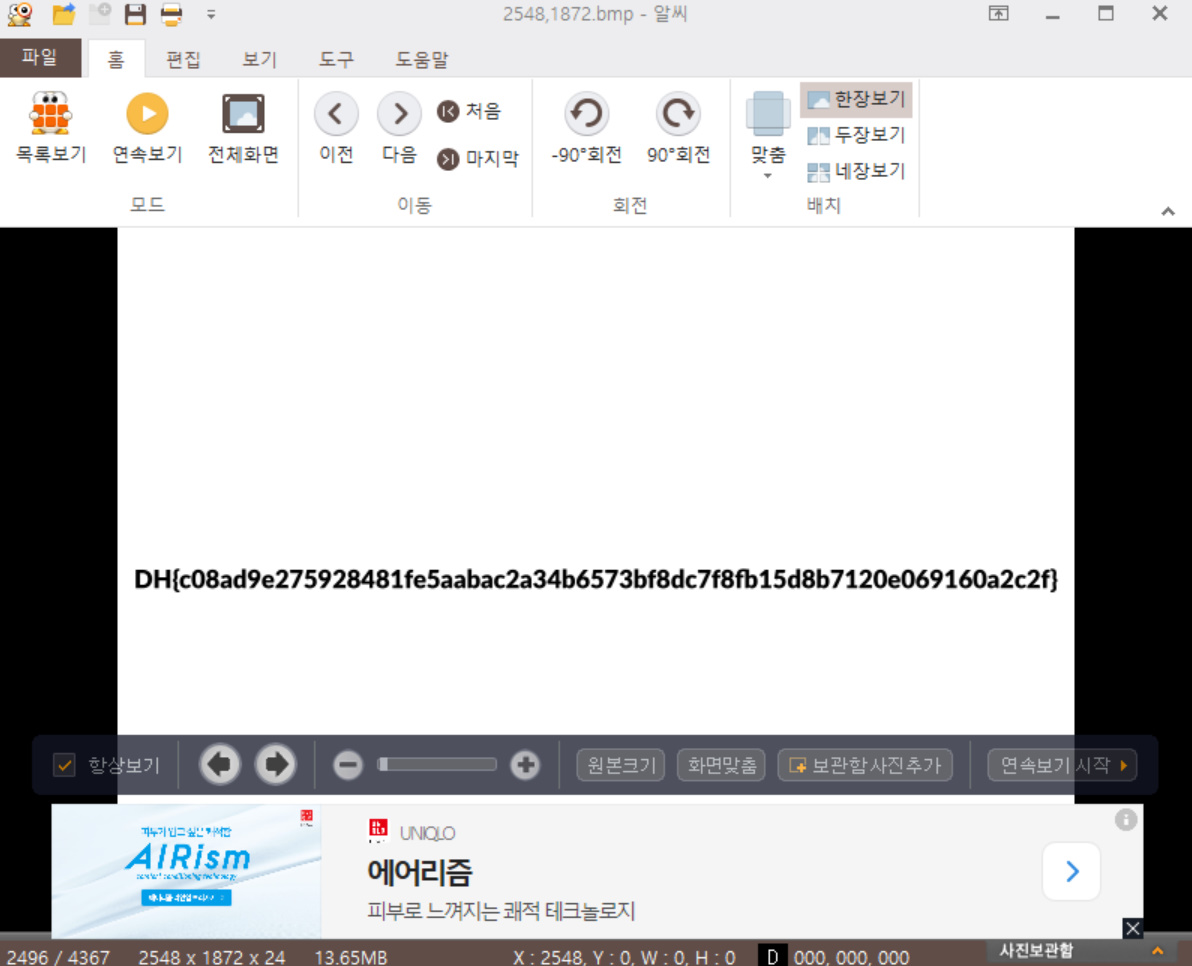


Figure 5 flag

# **Flag**

DH{c08ad9e275928481fe5aabac2a34b6573bf8dc7f8fb15d8b7120e069160a2c2f}

# **별도 첨부**

# **Reference**

* [URL]

<https://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2003_w/misc/bmp_file_format/bmp_file_format.htm>