

401 – Irresponsible predecessor

Team Information

Team Name : DogeCoin

Team Member : Dongbin Oh, Donghyun Kim, Donghyun Kim, Yeongwoong Kim

Email Address : dfc-dogecoin@naver.com

Instructions

Description. The new employee A asked the superior B for the access password of the old file server. B said, "It's a complicated password so I didn't memorize it. So, I saved the password in USB". They found the USB was in a warehouse, but it didn't mount. B said, "Maybe the USB is encrypted, but I don't remember the password. Repair the USB and access the file server!".

Target	Hash (MD5)
USB.dd	290a6c731938f3b6a234f219cf6be8e9

Questions

1. Repair the USB and then find password hint for the USB. (200 points)
2. What is the access password for the file server? (50 points)
3. Develop a tool to repair damaged parts of the given USB image file by analyzing the file system structure. (If library is required, include library or installation file) (150 points)

Tool requirements

- Input: USB.dd
- Output: Repaired USB image file

Teams must:

- Develop and document the step-by-step approach used to solve this problem to allow another examiner to replicate team actions and results.
- Specify all tools used in deriving the conclusion(s).

Tools used:

Name:	010editor	Publisher:	SweetScape Software
Version:	11.01		
URL:	https://www.sweetscape.com/010editor/		

Name:	APFS.bt	Publisher:	Yogesh Khatri
Version:	1.13		
URL:	https://www.sweetscape.com/010editor/repository/templates/file_info.php?file=APFS.bt&type=0&sort=		

Name:	Apfs2hashcat	Publisher:	Banaanhangwagen
Version:			
URL:	https://github.com/Banaanhangwagen/apfs2hashcat		

Name:	FTK Imager	Publisher:	AccessData
Version:	4.5		
URL:	https://accessdata.com/product-download/ftk-imager		

Name:	Mac-apt	Publisher:	Yogesh Khatri
Version:	1.4		
URL:	https://github.com/ydkhatri/mac_apt		

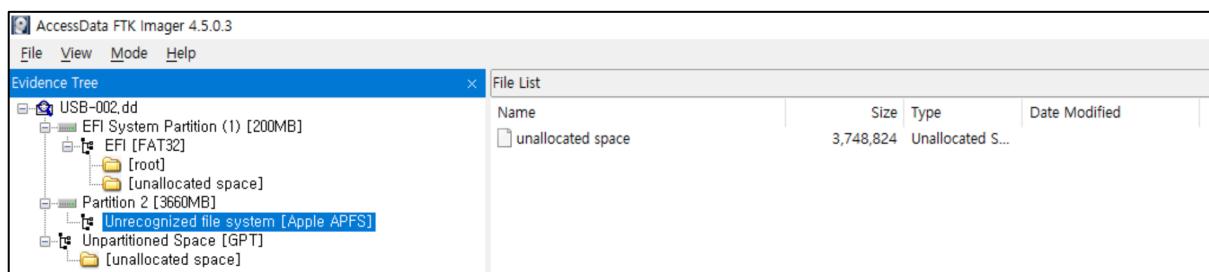
Name:	hashcat	Publisher:	hashcat
Version:	6.2.3		
URL:	https://hashcat.net/hashcat/		

Step-by-step methodology:

1. Repair the USB and then find password hint for the USB. (200 points)

문제 풀이를 진행하기 전에 문제의 조건으로 주어진 “Repair USB”의 범위를 설정하여야한다. 본 문제의 Description 상의 old file server의 password를 복구할 수 있는 것이 이 문제의 최종 목표이므로, 해당 Task를 달성할 수 있는 정도로 USB repair의 목표를 두고 분석을 진행하였다. 또한 문제 내에 USB가 암호화되어있다는 표현이 존재하므로 암호를 복호화 가능하도록 Repair 해야할 것이다.

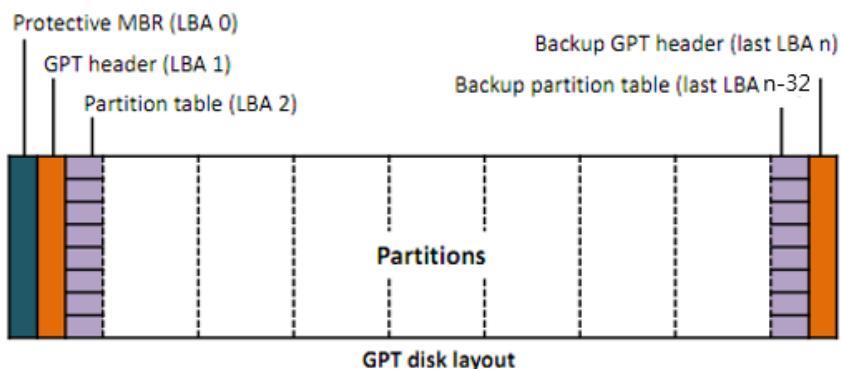
우선 주어진 dd image를 FTK Imager를 이용하여 확인하여 아래와 같은 결과를 얻을 수 있었다.



[그림 1] FTK Imager로 확인한 이미지 파일 정보

FTK Imager는 현재 USB-002.dd라는 파일은 GPT 구조를 가지고 있으며 FAT32의 EFI 영역과 Apple APFS로 추정되는 Partition 2를 확인할 수 있었다. 인식된 이름과 File List를 통해 현재 이 USB는 인식이 불가능하다는 문제가 있으므로 이를 해결하여야 한다.

① GPT 영역 검증 및 수정



[그림 2] GPT 구조

일반적인 GPT 의 구조는 위의 그림과 같이 LBA(Logical block addressing) 기준 0 번에 해당하는 영역에 Proactive MBR 이 존재하고, LBA 1에 GPT Header, LBA 2에 디스크 내에 존재하는 각 파티션의 정보가 담겨있는 Partition Table 이 존재한다. 또한 GPT 구조의 마지막에는 Backup partition table 과 Backup GPT Header 가 각각 존재한다. 문제에서 제공한 USB-002.dd 파일의 GPT 구조를 파악하기 위해

010Editor 를 이용해 GPT 구조와 APFS 의 구조가 사전에 정의되어 있는 APFS.bt Template 를 이용해 GPT 구조를 파악하였다. 실행 결과는 다음과 같다.

Name	Value	Start	Size	Color	Comment
struct GPT gpt_header		200h	200h	Fg: B	
char SIGNATURE[8]	EFI PART	200h	8h	Fg: B	
DWORD Revision	65536	208h	4h	Fg: B	
DWORD Headersize	92	20Ch	4h	Fg: B	
DWORD CRC32OfHeader	2945450342	210h	4h	Fg: B	
DWORD Reserved	0	214h	4h	Fg: B	
uint64 CurrentLBA	1	218h	8h	Fg: B	
uint64 BackupLBA	7907327	220h	8h	Fg: B	
uint64 FirstUsableLBA	34	228h	8h	Fg: B	
uint64 LastUsableLBA	7907294	230h	8h	Fg: B	
struct GUID DiskGUID	{991a0ddd-8ea3-4666-ae01-a25fc732f81b}	238h	10h	Fg: B	
uint64 PartitionEntriesLBA	2	248h	8h	Fg: B	
DWORD NumOfPartitionEntries	128	250h	4h	Fg: B	
DWORD SizeOfPartitionEntry	128	254h	4h	Fg: B	
DWORD CRC32ofPartitionArray	1081837668	258h	4h	Fg: B	
struct GPTPARTITIONTABLE table[128]		400h	4000h	Fg: B	
struct GPTPARTITIONTABLE table[0]		400h	80h	Fg: B	
struct GUID PartitionTypeGUID	{c12a7328-f81f-11d2-ba4b-00a0c93ec93b}	400h	10h	Fg: B	
struct GUID PartitionGUID	{0eb6dfc3-b31f-4a83-b324-9b27ab36d8c9}	410h	10h	Fg: B	
uint64 PartitionStartLBA	40	420h	8h	Fg: B	
uint64 PartitionEndLBA	409639	428h	8h	Fg: B	
uint64 PartitionProperty	0	430h	8h	Fg: B	
wchar_t PartitionName[36]	EFI System Partition	438h	48h	Fg: B	
struct GPTPARTITIONTABLE table[1]		480h	80h	Fg: B	
struct GUID PartitionTypeGUID	{7c3457ef-0000-11aa-aa11-00306543ecac}	480h	10h	Fg: B	
struct GUID PartitionGUID	{49e7ca7e-ce30-44ab-8f65-ec4360547dbd}	490h	10h	Fg: B	
uint64 PartitionStartLBA	409640	4A0h	8h	Fg: B	
uint64 PartitionEndLBA	7907287	4A8h	8h	Fg: B	
uint64 PartitionProperty	0	4B0h	8h	Fg: B	
wchar_t PartitionName[36]		4B8h	48h	Fg: B	

[그림 3] 010Editor 템플릿을 적용하여 확인한 GPT 데이터

기 FTK Imager 를 통해 확인할 수 있는 EFI 관련(GPTPARTITION TABLE table[0])외에, APFS 를 사용하고 있는 것으로 추정한 GPTPARTITION TABLE table[1]의 특징은 아래의 표와 같다.

[표 1] 주요 GPT 데이터

Key	Value
struct GUID PartitionTypeGUID	{7c3457ef-0000-11aa-aa11-00306543ecac}
struct GUID PartitionGUID	{49e7ca7e-ce30-44ab-8f65-ec4360547dbd}
uint64 PartitionStartLBA	409640
uint64 PartitionEndLBA	7907287
uint64 PartitionProperty	0
wchar_t PartitionName[36]	

이 데이터 중 의미있는 데이터는 Partition Type GUID 이다. Wikipedia 의 Partition GUID 를 정리해둔 웹 페이지에 따르면¹ 해당 GUID 값은 Apple APFS container, APFS FileVault volume container 로 알려져 있다. 이를 통해 FTK Imager 가 APFS 를 사용하고 있는 파티션이라고 해석한 기준을 파악할 수 있다.

¹ https://en.wikipedia.org/wiki/GUID_Partition_Table

또한 PartitionStartLBA 를 통해 Partition 시작 위치를 알 수 있고, PartitionEndLBA 를 통해 Partition 끝 위치를 알 수 있고, 이 정보들을 통해 Partition 의 크기를 확인할 수 있다.

추가적으로 Backup GPT header 와 Backup Partition Table 을 조사하였으나 의미있는 데이터를 얻을 수는 없었다.

② APFS 구조

APFS의 구조는 APFS라는 Container 안에 데이터를 담고 있는 Partition이 존재하는 구조이다. Container 정보의 경우 Main Superblock 과 Checkpoint superblock에 포함되어 있고, Partition 정보의 경우 Volume Superblock에 포함되어 있다. 이후 Volume Superblock에 기록된 B-Tree 구조에 따라 Root부터 File 에 대한 데이터들이 기록되어 있다.

이하의 APFS 구조는 Decoding the APFS file system² 과 APFS INTERNALS FOR FORENSIC ANALYSIS Whitepaper³를 주로 참고하였고, Repair 방법은 Forensic APFS File Recovery⁴를 참고하였다. 뿐만 아니라 전체적인 APFS 의 동작은 Apple File System Reference⁵를 참고 하였다.

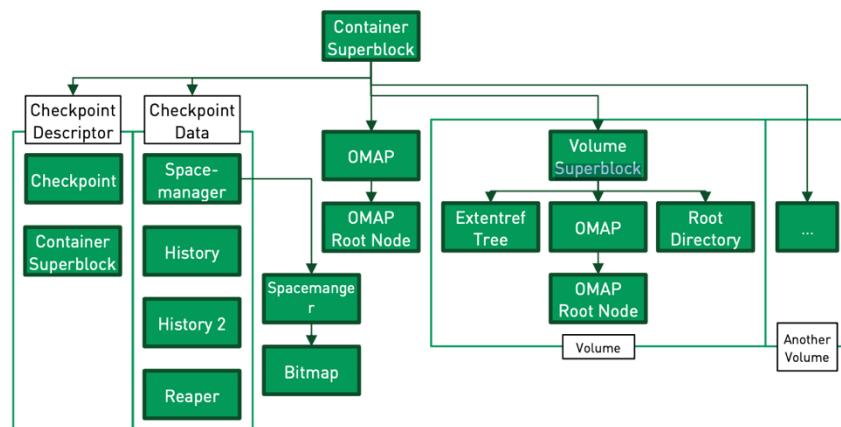


Figure 3: APFS composition

【그림 4】 APFS 구조

본 문제에서 APFS 의 Mounting과 Encryption 관련된 기능을 Repair 하므로 Apple File System Reference의 “Mounting an Apple File System Partition” 과 “Accessing Encrypted Objects”를 참고 하였다. 아래는 각각의 절차에 관한 내용이다.

² Hansen, Kurt H., and Fergus Toolan. "Decoding the APFS file system." Digital Investigation 22 (2017): 107-132.

³ Dewald, Ing Andreas. "ERNW WHITEPAPER 58." (2017).

⁴ Plum, Jonas, and Andreas Dewald. "Forensic apfs file recovery." Proceedings of the 13th International Conference on Availability, Reliability and Security. 2018.

⁵ <https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf>

1. Read block zero of the partition. This block contains a copy of the container superblock (an instance of `fnx_superblock_t`). It might be a copy of the latest version or an old version, depending on whether the drive was unmounted cleanly.
2. Use the block-zero copy of the container superblock to locate the checkpoint descriptor area by reading the `nx_xp_desc_base` field.
3. Read the entries in the checkpoint descriptor area, which are instances of `checkpoint_map_phys_t` or `nx_superblock_t`.
4. Find the container superblock that has the largest transaction identifier and isn't malformed. For example, confirm that its magic number and checksum are valid. That superblock and its checkpoint-mapping blocks comprise the latest valid checkpoint. The superblock's fields, like `nx_xp_desc_blocks` and `nx_data_len`, indicate which checkpoint-mapping blocks belong to that superblock.
5. Read the ephemeral objects listed in the checkpoint from the checkpoint data area into memory. If any of the ephemeral objects is malformed, the checkpoint that contains that object is malformed; go back to the previous step and mount from an older checkpoint.
6. Locate the container object map using the `nx_omap_oid` field of the container superblock.
7. Read the list of volumes from the `nx_fs_oid` field of the container superblock. If you're mounting only a particular volume, you can ignore the virtual object identifiers for the other volumes.
8. For each volume, look up the specified virtual object identifier in the container object map to locate the volume superblock (an instance of `apfs_superblock_t`). If you're mounting only a particular volume, you can skip this step for the other volumes.
9. For each volume, read the root file system tree's virtual object identifier from the `apfs_root_tree_oid` field, and then look it up in the volume object map indicated by the `apfs_omap_oid` field. If you're mounting only a particular volume, you can skip this step for the other volumes.
10. Walk the root file system tree as needed by your implementation to mount the file system.

[그림 5] APFS 마운트 과정

1. Locate the container's keybag using the `nx_keylocker` field of `nx_superblock_t`.
2. Unwrap the container's keybag using the container's UUID, according to the algorithm described in RFC 3394.
3. Find an entry in the container's keybag whose UUID matches the volume's UUID and whose tag is `KB_TAG_VOLUME_KEY`. The key data for that entry is the wrapped VEK for this volume.
4. Find an entry in the container's keybag whose UUID matches the volume's UUID and whose tag is `KB_TAG_VOLUME_UNLOCK_RECORDS`. The key data for that entry is the location of the volume's keybag.
5. Unwrap the volume's keybag using the volume's UUID according to the algorithm described in RFC 3394.
6. Find an entry in the volume's keybag whose UUID matches the user's Open Directory UUID and whose tag is `KB_TAG_VOLUME_UNLOCK_RECORDS`. The key data for that entry is the wrapped KEK for this volume.
7. Unwrap the KEK using the user's password, and then unwrap the VEK using the KEK, both according to the algorithm described in RFC 3394.

[그림 6] APFS 암호화 오브젝트 접근 과정

③ Checkpoint Superblock을 통한 Main Superblock 설정

FTK Imager와 Partition GUID를 통해 조사 대상인 파티션이 APFS를 사용한다는 것을 추정할 수 있었다.

이후 분석을 위해 010Editor의 APFS.bt 템플릿을 활용하여 APFS 구조 분석을 진행하였다. APFS.bt 템플릿 실행 결과, 우선 아래의 그림과 같은 에러 메세지를 확인할 수 있었다.

```
Output
Executing template 'C:\Users\sin90\Desktop\APFS.bt' on 'C:\Users\sin90\Desktop\USB-002.dd'...
Error, starting point not an APFS container superblock. Set the 'Apfs_Offset' variable to correct value!Result = 1 [lh]
```

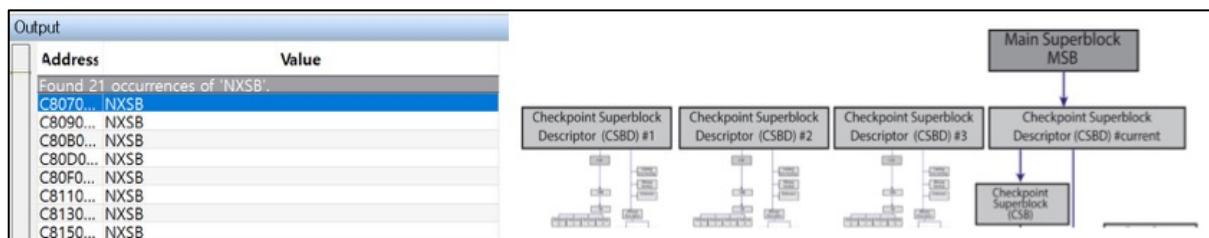
[그림 7] 010Editor 템플릿 적용 시, 발생 에러

GPT의 PartitionStartLBA 409640 위치를 확인해보니 Main Superblock의 데이터가 존재해야 할 영역이 0x00으로 덮여있는 것을 확인할 수 있었다. 이를 통해 Main Superblock의 데이터를 복구해야 할 필요성이 있는 것을 파악할 수 있다.

Startup	USB-002_test_GPTmodified2_4316_uuidchange_4318_volumid_checksum_rootnode.dmg	USB-002.dd
	0 1 2 3 4 5 6 7 8 9 A B C D E F	0123456789ABCDEF
C80:5000h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5010h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5020h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5030h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5040h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5050h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5060h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5070h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5080h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5090h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

[그림 8] 0x00으로 덮여있는 Main Superblock

Main Superblock이 누락된 APFS는 Checkpoint Superblock을 통해 특정 시점의 Main Superblock 데이터를 사용할 수 있다.⁶ Checkpoint Superblock을 확인하기 위해서는 “NXSB”라는 Signature를 사용할 수 있다. Signature 기반 검색 결과 disk image 내에 총 NXSB가 존재한다는 21개의 결과를 얻을 수 있었다.



[그림 9] NXSB Signature 검색 결과

Main Superblock의 경우 가장 최근의 데이터가 가장 마지막에 존재한다고 알려져 있으므로 현재 검색된 NXSB 시그니처를 기준으로 가장 마지막 NXSB를 기준으로 -0x20 위치부터 0x1000에 해당하는 길이만큼을 PartitionStartLBA 409640에 복사하였다.

⁶ van der Slik, Maarten. "Research Project 1 APFS checkpoint management behaviour in macOS."

C83:1000h:	E6 C5 3C 6B 9C 4B E7 68 01 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	æÀ<köKçh.....
C83:1010h:	6E 00 00 00 00 00 00 00 01 00 00 80 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	n.....€.....
C83:1020h:	4E 58 53 42 00 10 00 00 F6 4C 0E 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	NXSB....öL.....
C83:1030h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C83:1040h:	02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C83:1050h:	00 00 00 00 00 00 00 00 15 04 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C83:1060h:	6F 00 00 00 00 00 00 00 2C 00 00 00 84 0E 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	o.....,.....
C83:1070h:	01 00 00 00 00 00 00 00 2D 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-
C83:1080h:	00 00 00 00 39 02 00 00 2A 00 00 00 02 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 009...*

C80:5000h:	E6 C5 3C 6B 9C 4B E7 68 01 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	æÀ<köKçh.....
C80:5010h:	6E 00 00 00 00 00 00 00 01 00 00 80 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	n.....€.....
C80:5020h:	4E 58 53 42 00 10 00 00 F6 4C 0E 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	NXSB....öL.....
C80:5030h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5040h:	02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5050h:	00 00 00 00 00 00 00 00 15 04 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C80:5060h:	6F 00 00 00 00 00 00 00 2C 00 00 00 84 0E 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	o.....,.....
C80:5070h:	01 00 00 00 00 00 00 00 2D 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00-

[그림 10] Checkpoint Superblock 데이터 복사

이후 APFS Template를 재실행하면 아래처럼 APFS의 Container Block0 Parsing 된 것을 확인할 수 있다.

struct obj_header hdr		C80500...20h	Fg: B
uint64 cksum	7559093634750793190	C80500... 8h	Fg: B
uint64 oid	1	C80500... 8h	Fg: B
uint64 xid	110	C80501... 8h	Fg: B
enum obj_type type	obj_type_container_superblock (1)	C80501... 2h	Fg: B
ushort flags	OBJ_EPHEMERAL,	C80501... 2h	Fg: B
enum obj_type subtype	0	C80501... 2h	Fg: B
struct container_superblock body		C80502... 548h	Fg: B
char magic[4]	NXSB	C80502... 4h	Fg: B
uint block_size	4096	C80502... 4h	Fg: B
uint64 block_count	937206	C80502... 8h	Fg: B
uint64 features_0	0	C80503... 8h	Fg: B
uint64 read_only_compatible_features	0	C80503... 8h	Fg: B
uint64 incompatible_features	2	C80504... 8h	Fg: B
struct Uuid uuid	00000000-0000-0000-0000-000000000000	C80504... 10h	Fg: B
uint64 next_oid	1045	C80505... 8h	Fg: B
uint64 next_xid	111	C80506... 8h	Fg: B
uint xp_desc_blocks	44	C80506... 4h	Fg: B

[그림 11] 정상적으로 Parsing된 Container Block

0| 중 유의하여 볼 데이터는 다음과 같다.

[표 2] Parsing된 Container Block 정보

Key	Value
struct Uuid uuid	00000000-0000-0000-0000-000000000000
uint xp_desc_blocks	44
uint64 xp_desc_base	1
uint xp_desc_next	0
uint xp_desc_index	42
struct obj omap	4317
uint64 flags	NX_CRYPTO_SW

uint64 keylocker_paddr	4208
------------------------	------

현재 데이터를 보면 uuid 값이 00000000-0000-0000-0000-000000000000 이다. Uuid의 경우 “Accessing Encrypted Objects”에서 복호화를 위해 사용되는 항목으로 위의 데이터와 같은 값들이 나올 수 없다. 이를 위해 Checkpoint Superblock에서 uuid 값이 존재하는 것을 확인하였다. 해당 영역에는 NXSB 시그니처가 존재하지 않는 점도 확인할 수 있었다. 이는 xp_desc_index를 통해 확인된 checkpoint이다.

▼struct obj_checkpoint_desc_nx[21]	C81B00... 1000h	Fg: B(error)
> struct obj_header hdr	C81B00... 20h	Fg: B
▼struct container_superblock body	C81B02... 548h	Fg: B
> char magic[4]	C81B02... 4h	Fg: B
uint block_size	4096	C81B02... 4h
uint64 block_count	937206	C81B02... 8h
uint64 features_0	0	C81B03... 8h
uint64 read_only_compatible_features	0	C81B03... 8h
uint64 incompatible_features	2	C81B04... 8h
> struct Uuid uuid	095746c7-76e1-4400-b4d9-ea614bd9ff3f	C81B04... 10h

[그림 12] UUID 값 및 시그니처 확인

해당 값을 Main Superblock에 복사 후, Checksum을 계산하여 Main Superblock을 업데이트하였다. Checksum 계산은 위해 아래의 Python Source Code를 이용하였다.

[표 3] Checksum 계산 코드

```
import numpy as np

def create_checksum(data):
    sum1 = np.uint64(0)
    sum2 = np.uint64(0)

    mod_value = np.uint64(4294967295) # 2<<31 - 1

    for i in range(int(len(data) / 4)):
        dtype = np.dtype(np.uint32)
        dtype = dtype.newbyteorder('L')
        value = np.frombuffer(data[i * 4:(i + 1) * 4], dtype=dtype)

        sum1 = (sum1 + np.uint64(value)) % mod_value
        sum2 = (sum2 + sum1) % mod_value

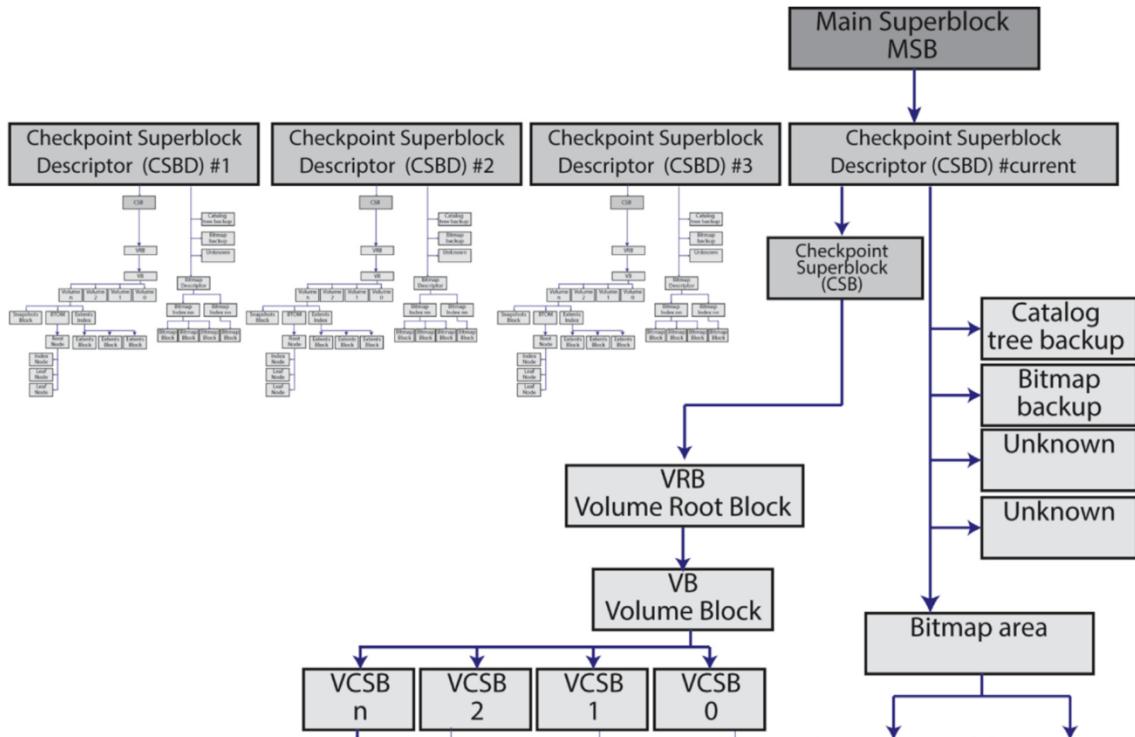
    check1 = mod_value - ((sum1 + sum2) % mod_value)
    check2 = mod_value - ((sum1 + check1) % mod_value)

    return (check2 << 32) | check1

def check_checksum(data):
    dtype = np.dtype(np.uint64)
    dtype = dtype.newbyteorder('L')
    print("original checksum : %s", np.frombuffer(data[:8], dtype=dtype))
    print("calculated checksum : %s", create_checksum(data[8:]))
    print(np.frombuffer(data[:8], dtype=dtype) == create_checksum(data[8:]))
    return (np.frombuffer(data[:8], dtype=dtype) == create_checksum(data[8:]))[0]

if __name__ == '__main__':
    with open('checksum_hex.dd', 'rb') as io:
        dump = io.read()
    check_checksum(dump)
```

④ Volume Signature 기반 Volume Superblock 탐색 및 설정



[그림 13] Volume Superblock 탐색

Volume Superblock을 찾기 위해서는 Main Superblock에 정의된 omap_oid를 따라가면 된다. oid의 경우 referencing object를 나타내게 되며 xid의 경우에는 Object의 version을 나타내게된다. APFS Template에서 각 구조체 별 데이터 형식을 미리 입력해두었기 때문에 아래와 같이 omap_oid -> omap -> b_omap body -> tree_oid -> tree -> node body -> omap_node_entry entries -> omap_val val -> paddr -> omap을 따라가면 정상적인 APFS 파일 시스템이면, Volume Superblock(Volume Root Block)이 나타나게 된다. 아래 그림의 좌측이 정상적인 APFS의 경우이고 우측이 현재 USB-002.dd의 이미지이다.

우측의 경우 APFS의 volume superblock의 oid 값을 나타내는 paddr이 68719476752으로 나타내있다. 해당 값은 현재 APFS Container의 크기를 생각해보았을 때, 불가능하며 해당 값을 수정해야한다.

struct omap_node_entry entries	OID 1026, XID 110	D8E3... 4h	Fc	struct omap_node_entry entries	OID 1026, XID 110	D8E3... 4h	Fc
short key_offset	16	D8E3... 2h	Fc	short key_offset	16		
short data_offset	32	D8E3... 2h	Fc	short data_offset	32		
> struct omap_key key_header		D8E3... 10h	Fc	> struct omap_key key_header		D8E3... 10h	Fc
struct omap_val val		D8E3... 10h	Fc	struct omap_val val		D8E3... 10h	Fc
uint flags	(16) CRYPTO_GENERATION	D8E3... 4h	Fc	uint flags	(16) CRYPTO_GENERATION	D8E3... 4h	Fc
uint size	16	D8E3... 4h	Fc	uint size	16	D8E3... 4h	Fc
uint64 paddr	4316	D8E3... 8h	Fc	uint64 paddr	68719476752	D8E1... 400h	Fc
> struct obj_omap		D8E1... 1000h	Fc				
> struct obj_header hdr		D8E1... 20h	Fc				
struct apfs_superblock body		D8E1... 400h	Fc				

[그림 14] paddr 값 수정

Main Superblock이 “NXSB”라는 Signature를 가진 것처럼, Volume Superblock의 경우에는 “APSB”라는 Signature를 가지고 있다. 동일한 방법으로 APSB라는 Signature를 기반으로 USB-002.dd를 검색해본 결과 총 89개로 나타났다.

Address	Value
Found 89 occurrences of 'APSB'.	
D72E020h	APSB
D73E020h	APSB
D745020h	APSB
D752020h	APSB
D765020h	APSB
D76D020h	APSB
D773020h	APSB
D77C020h	APSB
D780020h	APSB

[그림 15] APSB Signature 검색

Volume Superblock을 가르키는 node body의 entry count 가 1 인 것으로 보아 실질적인 Volume은 1 개이고, 89개의 APSB 중 적합한 offset을 선택하여 Volume을 인식하여야한다. 적절한 offset을 선정하기 위해 Volume Superblock의 구조에서 아래의 표와 같이 의미 있는 데이터를 선별하였다.

[표 4] Volume Superblock 데이터

Key	Offset	Description
Checksum	0-7	Checksum
Oid	8-15	Volume Superblock 의 reference oid
Xid	16-23	Volume Superblock 의 reference version
Signature(APSB)	32-35	Volume Superblock 시그니처
Omap_oid	128-135	Volume superblock 의 omap
Root_map_oid	136-143	B-tree 의 Root 가 해당하는 OMAP oid
Vol_uuid	240-255	Volume 자체의 uuid
Last_mod_time	256-263	Volume 의 최종 수정 시간
volume_flags vol_flags	264-271	Volume 의 특징을 나타내는 값

위의 정보 중, Oid, Xid, Signature 값은 Container Superblock으로부터 refer된 1026, 110, APSB 값으로 고정이므로 Volume Superblock의 offset을 찾는데 사용할 수 있다. 뿐만 아니라, 문제의 지문으로부터 Volume이 encrypt 되어있고, 가장 최근에 사용한 Volume Superblock의 경우 Last_Mod_time 값이 가장 클 것이므로 위의 기준을 근거로 python code를 작성해 Carving을 시도하였다. 관련 소스 코드는 아래와 같다.

[표 5] Volume Superblock Carving 코드

```
import numpy as np

def create_checksum(data):
    sum1 = np.uint64(0)
    sum2 = np.uint64(0)

    mod_value = np.uint64(4294967295) # 2<<31 - 1

    for i in range(int(len(data)) / 4):
        dtype = np.dtype(np.uint32)
        dtype = dtype.newbyteorder('L')
        value = np.frombuffer(data[i * 4:(i + 1) * 4], dtype=dtype)

        sum1 = (sum1 + np.uint64(value)) % mod_value
        sum2 = (sum2 + sum1) % mod_value

    check1 = mod_value - ((sum1 + sum2) % mod_value)
```

```

check2 = mod_value - ((sum1 + check1) % mod_value)

return (check2 << 32) | check1

def check_checksum(data):
    dtype = np.dtype(np.uint64)
    dtype = dtype.newbyteorder('L')
    print("original checksum : %s", np.frombuffer(data[:8], dtype=dtype))
    print("calculated checksum : %s", create_checksum(data[8:]))
    print(np.frombuffer(data[:8], dtype=dtype) == create_checksum(data[8:]))
    return (np.frombuffer(data[:8], dtype=dtype) == create_checksum(data[8:]))[0]

def validate_volume_superblock(data):
    dtype = np.dtype(np.uint64)
    dtype = dtype.newbyteorder('L')
    dumpstart = [0]
    real_encrypted = []
    modified_times = []
    checksum_result = []
    for i in dumpstart:
        results = dump.find(b'APSB',dumpstart[-1]+1)
        apsb_oid = dump[results-24 : results-16]
        checksum_data = dump[results-32 : results+4064]
        if results > 0 and apsb_oid == b'\x02\x04\x00\x00\x00\x00\x00\x00':
            dumpstart.append(results)
            if dump[results+232 : results+240] == b'\x08\x00\x00\x00\x00\x00\x00\x00':
                print("#####")
                print("hex offset : %s", results)
                print("#####")
                print('this is encrypted checkpoint volumne superblock')
                print("omap_oid %s ", dump[results+96 : results+104])
                print("root_omap_oid %s ", dump[results+104 : results+112])
                print("volume_name %s", dump[results+672 : results+672+256])
                print("modified time %s", np.frombuffer(dump[results+224 : results+232], dtype=dtype) )
                real_encrypted.append(results)
                modified_times.append(np.frombuffer(dump[results+224 : results+232], dtype=dtype))
                checksum_result.append(check_checksum(checksum_data))

    print(modified_times)
    print(real_encrypted)
    max_time = max(modified_times)
    max_time_index =[i for i, j in enumerate(modified_times) if j == max_time]
    print("!!!!!!!!!!!!!!")

    for i in range(0,len(real_encrypted)):
        print("final offset %s", real_encrypted[i])
        block_num=(realEncrypted[i]-32 - (409640 * 512)) / 4096
        print("block number = %s", block_num)
        print("checksum results = %s ",checksum_result[i] )

if __name__ == '__main__':
    with open('USB-002.dd', 'rb') as io:
        dump = io.read()
        validate_volume_superblock(dump)

```

실행 결과는 다음의 그림과 같다.

```
!!!!!!  
final offset %s 226897952  
block number = %s 4190.0  
checksum results = %s True  
final offset %s 227029024  
block number = %s 4222.0  
checksum results = %s True  
final offset %s 227074080  
block number = %s 4233.0  
checksum results = %s True  
final offset %s 227182752  
block number = %s 4240.0  
checksum results = %s True  
final offset %s 227127328  
block number = %s 4246.0  
checksum results = %s True  
final offset %s 227192864  
block number = %s 4262.0  
checksum results = %s True  
final offset %s 227246112  
block number = %s 4275.0  
checksum results = %s True  
final offset %s 227287072  
block number = %s 4285.0  
checksum results = %s True  
final offset %s 227401760  
block number = %s 4313.0  
checksum results = %s True  
final offset %s 227414048  
block number = %s 4316.0  
checksum results = %s False
```

[그림 16] Volume Superblock Carving 코드

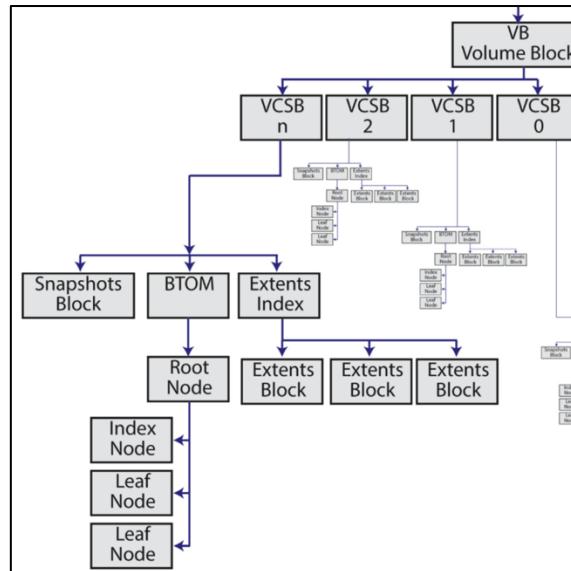
Oid, xid, signature, encrypt 조건을 만족하고 last mod time이 가장 큰 값을 가지고 있는 것은 10개이며 그 중 1개 Block number 4316이라고 표기된 것에서만 Checksum 비교 결과 False로 나타난 것을 확인할 수 있었다. Checksum Result가 False인 것으로 보아 해당 Volume signature가 문제가 있고, Repair가 필요 한 것으로 유추할 수 있다. Repair를 위해 omap_val의 paddr 을 4316으로 변경 후 Template를 재실행하였고 다음과 같이 Volume Superblock이 인식된 것을 확인할 수 있었다.

▼struct omap_val val		int64 apfs_total_blocks_freed	489
uint flags	(16) CRYPTO_GENERATION	struct Uuid vol_uuid	00000000-0000-0000-0000-000000000000
uint size	16	int64 last_mod_time	07/15/2021 08:06:09
uint64 paddr	4316	num volume_flags vol_flags	(8) onekey,
▼struct obj omap		struct apfs_modified_by formatted...	
► struct obj_header hdr		struct apfs_modified_by modified...	
▼struct apfs_superblock body		char volname[256]	DataBackup
► char magic[4]	APSB	int32 next_doc_id	3
uint fs_index	0	int16 apfs_role	(0) NONE
enum vol_features features	(2) hardlink_map_records,	int16 reserved	0

[그림 17] Volume Superblock 인식 결과

Volume Superblock 확인 결과 Container의 uuid와 동일하게 00000000-0000-0000-0000-000000000000으로 되어있어 추후 복구가 필요하다는 것을 예상할 수 있고, Volume의 이름이 DataBackup이라는 사실도 파악할 수 있다.

⑤ Root OMAP 영역 관련 복구



[그림 18] Root OMAP 영역 접근

Volume Superblock에서 실제 파일 데이터가 저장된 Root B-tree로 진입하기 위해서는 APFS 010Editor에 정의된 Template을 기준으로 apfs_superblock body -> omap_oid -> omap -> b_omap body -> tree_oid -> tree -> node body -> omap_node_entry entries를 통해 접근할 수 있다. 그러나 현재의 상태는 이러한 omap_node_entry entries를 확인할 수 없다.

struct node body		struct btree_info btree_information		struct node body	
enum btree_node_flags node_type	(7) BTNODE_ROOT, 0	uint32 bt_flags	(18) SEQ	enum btree_node_flags nod...	(7) BTNODE_ROOT,B
ushort level	0	uint32 bt_node_size	4096	ushort level	0
uint entry_count	11	uint32 bt_key_size	16	uint entry_count	0
uint16 table_space_off	0	uint32 bt_val_size	16	uint16 table_space_off	0
uint16 table_space_len	448	uint32 bt_longest_key	16	uint16 table_space_len	448
uint16 free_space_off	192	uint32 bt_longest_val	16	uint16 free_space_off	192
uint16 free_space_len	3168	uint64 bt_key_count	11	uint16 free_space_len	3168
uint16 key_free_list_off	48	uint64 bt_node_count	1	uint16 key_free_list_off	48

[그림 19] key_count, entry_count 비교

USB-002.dd의 경우 btree_info btree_information의 key_count가 11로 나타났다. 정상적인 APFS라면 key_count가 11이면 node body의 entry_count도 동일하게 11로 나타나야하지만, USB-002.dd의 경우 0으로 나타나있다. 이를 key_count 값인 11로 복구하고 Template을 재실행하면 아래와 같이 omap_node_entry entries가 잘 인식된 것을 확인할 수 있다.

▼struct node body	
enum btree_node_flags nod...	(7) BTNODE_ROOT,BTNODE_LEAF,BTNODE_F...
ushort level	0
uint entry_count	11
uint16 table_space_off	0
uint16 table_space_len	448
uint16 free_space_off	192
uint16 free_space_len	3168
uint16 key_free_list_off	48
uint16 key_free_list_len	16
uint16 val_free_list_off	64
uint16 val_free_list_len	16
> struct omap_node_entry ent...	OID 1028, XID 109
> struct omap_node_entry ent...	OID 1030, XID 108
> struct omap_node_entry ent...	OID 1031, XID 107
> struct omap_node_entry ent...	OID 1033, XID 106
> struct omap_node_entry ent...	OID 1034, XID 109
> struct omap_node_entry ent...	OID 1035, XID 109
> struct omap_node_entry ent...	OID 1037, XID 106
> struct omap_node_entry ent...	OID 1038, XID 102
> struct omap_node_entry ent...	OID 1041, XID 102
> struct omap_node_entry ent...	OID 1042, XID 108
> struct omap_node_entry ent...	OID 1043, XID 109
> struct btree_info btree_infor...	

[그림 20] 정상적으로 인식된 omap_node_entry entries

⑥ APSB 시그니쳐 카빙을 통한 Volume UUID 값 복구

Volume UUID 값을 구하기 위해, 앞서 시도한 APSB 시그니처 카빙을 이용해 사용한 소스코드에 UUID를 Print 하는 코드를 추가하여 아래의 결과를 얻을 수 있었다. 현재 사용한 4316 APFS Volume Superblock 외에 Volume UUID가 동일한 ed26aaa3-f340-4aa1-b9c4-6294dc84adbf 값을 나타내고 있음을 확인할 수 있다. 이를 통해 00000000-0000-0000-000000000000 역시 ed26aaa3-f340-4aa1-b9c4-6294dc84adbf 값이어야함을 추정할 수 있고, 해당 값으로 업데이트 하였다.

[그림 21] Volume UUID 복구 결과

⑦ Checksum 확인 및 수정

현재까지 value 수정된 영역과 File system mounting, Encryption 관련된 영역에 대하여 각 Header에 존재하는 Checksum을 검증하였다. 검증 방법은 위의 Source code를 이용하였고 검증 대상은 아래와 같다.

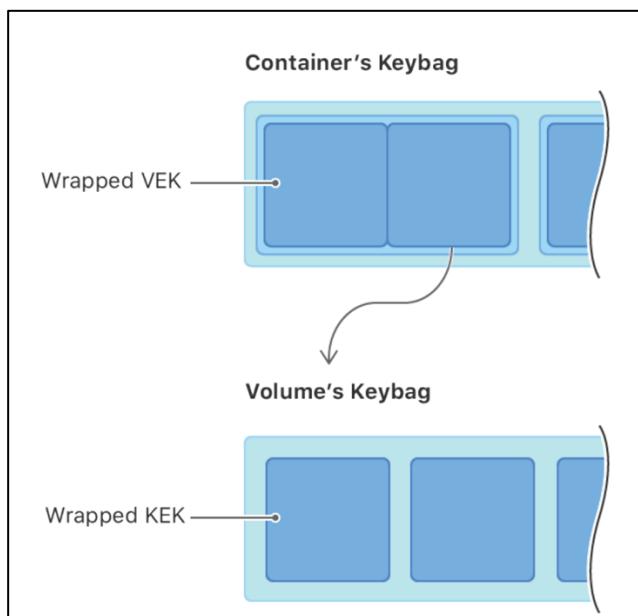
- Main Superblock (size 0x1000)
 - Volume Superblock (size 0x1000)
 - Main Superblock - omap (size 0x1000)
 - Main Superblock - omap body - tree (size 0x1000)
 - Main Superblock - omap body - tree - omap node entry entries - omap (size 0x1000)

<pre> uint64 tree_oid 4318 struct obj_tree struct obj_header hdr uint64 cksm 1178781647110256577 uint64 oid [base) sin90@odori-bin-iii-MacBookPro:~% uint64 xid original checksum .%s enum obj_type type calulated checksum :> ushort flags [False] </pre>	<pre> uint64 tree_oid 4318 struct obj_tree struct obj_header hdr uint64 cksm 117887749982307974 uint64 oid 4318 </pre>
---	--

[그림 22] Checksum 설정

그림과 같이 검증 대상을 기준으로 이상이 존재하는 checksum을 변경하여주었다.

⑧ 복호화 루틴을 이용한 검증 및 Password Hint 확인



[그림 23] APFS 암호화 구성

현재의 Main superblock 의 Container UUID 값과 Volume Superblock 의 Volume UUID 값은 이전 상태의 Checkpoint 로 부터 가져온 것이거나, snapshot 등의 데이터로부터 가져온 데이터들이다. 이에 대한 추가 검증이 필요할 뿐 아니라 복호화를 위해 UUID 이외의 데이터들이 다 setup 되었는지 확인할 필요가 있다.

APFS 의 암호화는 위의 그림과 같이 Container Keybag 안에 Wrapped VEK, Wrapped KEK 를 unwrap 하면 Volume Keybag 과 Wrapped KEK 가 존재한다. Wrapped KEK 를 decrypt 하면 이제 복호화에 필요한 decryption key 를 획득할 수 있다. 암호화 과정에서는 RFC 3394⁷를 따른다.

Apple File System Reference 의 Accessing Encrypted Objects 에 UUID 그리고 user password 와 관련된 부분을 아래와 같이 표기하였다.

[표 6] Accessing Encrypted Objects (UUID, User Password)

- 2. Unwrap the container's keybag using the **container's UUID**, according to the algorithm described in RFC 3394.

⁷ <https://datatracker.ietf.org/doc/rfc3394/>

3. Find an entry in the container's keybag whose UUID matches the **volume's UUID** and whose tag is KB_TAG_VOLUME_KEY. The key data for that entry is the wrapped VEK for this volume.
4. Find an entry in the container's keybag whose UUID matches the **volume's UUID** and whose tag is KB_TAG_VOLUME_UNLOCK_RECORDS. The key data for that entry is the location of the volume's keybag.
7. Unwrap the KEK using the **user's password**, and then unwrap the VEK using the KEK, both according to the algorithm described in RFC 3394.

Libfsapfs 의 개발자가 정리한 Documentation⁸을 통해 아래와 같이 UUID 및 Password Hint 의 구조와 같은 정보를 얻을 수 있었다.

8.4.1. Key bag entry header				8.4.2. Key bag entry types		
				Container key bag entry types		
Offset	Size	Value	Description	Value	Identifier	Description
0	16		Volume identifier (ke_uuid) Contains a UUID stored in big-endian	0x00	KB_TAG_UNKNOWN	Unknown
16	2		Entry type (ke_tag) See section: Key bag entry types	0x01	KB_TAG_WRAPPING_KEY	Wrapping key
18	2		Entry data size (ke_keylen)	0x02	KB_TAG_VOLUME_KEY	Volume master key See section: Key encrypted key (KEK) packed object
20	4		Unknown (padding)	0x03	KB_TAG_VOLUME_UNLOCK_RECORDS	Volume key bag extent See section: Key bag data extent
				0x04	KB_TAG_VOLUME_PASSPHRASE_HINT	Passphrase hint
				0xf8	KB_TAG_USER_PAYLOAD	Unknown (user payload)

The volume master key is encrypted with a volume key.

[그림 24] APFS Key bag entry header, Password Hint 정보

APFS 의 복호화 작업은 ydkhatri 의 mac_apt github repository⁹ 를 응용하여 수행하였다. 해당 repository 는 MAC 관련 아티팩트를 자동으로 수집해주는 도구이며 encrypt APFS 를 지원한다. 이 중 extract_apfs_fs.py¹⁰ 를 일부 수정하였다. 주요 수정 내용으로는 동일 repository 내에 decryptor 를 불러오도록 하고, decryptor.py 내에 EncryptedVol class 의 find_wrapped_vek 함수와 decrypter() 함수에 print 를 추가 하였다. 관련 소스코드 중 중요 부분은 아래와 같다.

[표 7] APFS 복호화 주요 코드

```
#Modified version of extract_apfs_fs.py
~ 중략 ~

def tryingdecryptor(img, container_size, container_start_offset, container_uuid):
    global mac_info
    from plugins.helpers import decryptor
    mac_info = macinfo.ApfsmacInfo(mac_info.output_params, mac_info.password, mac_info.dont_decrypt)
    mac_info.pytsk_image = img    # Must be populated
    mac_info.is_apfs = True
    mac_info.macos_partition_start_offset = container_start_offset # apfs container offset
```

⁸ [https://github.com/libyal/libfsapfs/blob/main/documentation/Apple%20File%20System%20\(APFS\).asciidoc](https://github.com/libyal/libfsapfs/blob/main/documentation/Apple%20File%20System%20(APFS).asciidoc)

⁹ https://github.com/ydkhatri/mac_apt

¹⁰ https://github.com/ydkhatri/mac_apt/blob/master/extract_apfs_fs.py

```

mac_info.apfs_container = ApfsContainer(img, container_size, container_start_offset)
print(container_size / 4096 )
print("len : %s", len(mac_info.apfs_container.volumes))
vol=mac_info.apfs_container.volumes[-1]
decryption_key = decryptor.EncryptedVol(vol, None, '7412207~').decryption_key
vol.encryption_key = decryption_key
apfs_parser = ApfsFileSystemParser(vol, None)
apfs_parser.read_volume_records()

```

~ 중략 ~

NXSB_location= 209735680

```

mac_info.pytsk_image = img
if IsApfsContainer(img, NXSB_location):
    uuid = GetApfsContainerUuid(img, NXSB_location)
    print('Found an APFS container with uuid: {}'.format(str(uuid).upper()))
    #ParseVolumesInApfsContainer(img, None, img.get_size(), 0, uuid)
    tryingdecryptor(img, img.get_size(), NXSB_location, uuid)
else: # perhaps this is a full disk image
    try:
        vol_info = pytsk3.Volume_Info(img)
        vs_info = vol_info.info # TSK_VS_INFO object
        mac_info.vol_info = vol_info
        # Try as a partition/container first
        ParsePartitionTables(img, vol_info, vs_info)
        Disk_Info(mac_info, args.input_path).Write()
    except Exception as ex:
        log.exception("Error while trying to read partitions on disk")

```

~ 중략 ~

#Modified version of decryptor.py

~ 중략 ~

```

def find_wrapped_vek(self, container_keybag):
    """
    :param container_keybag: The parsed container keybag
    :return: tuple (The wrapped VEK for the volume, vek UUID, enc_type)
    """

    print("Searching for VEK now")
    volume_uuid = self.ApfsVolume.uuid
    volume_uuid = volume_uuid.replace("-", "")
    print(volume_uuid)

    for kl_entry in container_keybag.mk_locker.kl_entries:
        if kl_entry.KeyBag_Tags == KB_TAG_VOLUME_KEY:
            print("match!")
            # Converts the UUID in the unwrapped keybag entry to GUID like format for comparisons
            byte_uuid = kl_entry.UUID
            readable_UUID = convert_keybag_uuid_to_string(byte_uuid)
            print('this is keybag volume uuid %s', readable_UUID)
            if readable_UUID == volume_uuid:
                print("Found a UUID within the kl_entry with a Tag of two and a UUID that matches the "
                      "Volume UUID we are trying to decrypt!")
                return kl_entry.blob_header.bag_data, kl_entry.blob_header.vek_uuid, kl_entry.blob_header.enc_type

```

```

return None, None

~ 종략 ~

def decrypter(self):

    # Uses the get_wrapped_keybag() function to get the wrapped keybag from the offsets defined in the
    # Container Super Block
    # THIS IS STEP 1 OF THE APFS ACCESSING ENCRYPTED OBJECTS DOCUMENTATION
    wrapped_container_keybag = self.get_wrapped_keybag(self.wrapped_keybag_offset, self.keylocker_block_count)
    unwrapped_container_keybag = self.decrypt_keybag(wrapped_container_keybag, self.wrapped_keybag_offset, self.uuid)
    print("Successfully unwrapped the Wrapped Keybag!")
    print("unwrapped container keybag %s",unwrapped_container_keybag)
    #print('unwrapped_container_keybag : %s', unwrapped_container_keybag)
    parsed_container_keybag = parse_vek_keybag(unwrapped_container_keybag)

    # Parses the unwrapped kb_locker object derived from the wrapped keybag into structures
    # THIS IS STEP 3 OF THE APFS ACCESSING ENCRYPTED OBJECTS DOCUMENTATION
    print("Parsing the unwrapped keybag now")
    wrapped_vek, vek_uuid, vek_enc_type = self.find_wrapped_vek(parsed_container_keybag)

    # Returns if no UUID is found
    if wrapped_vek is None:
        print("COULD NOT FIND MATCHING UUID IN KEYBAG (for obtaining wrapped VEK). VOLUME CANNOT BE
DECRYPTED!!")
        return

    # Finds the volumes keybag by looking at the unwrapped containers keybag with a tag of
    # KB_TAG_VOLUME_UNLOCK_RECORDS
    # THIS IS STEP 4 OF THE APFS ACCESSING ENCRYPTED OBJECTS DOCUMENTATION
    volume_keybag_start_addr, volume_keybag_block_count, volume_uuid =
    self.find_volume_keybag_details(parsed_container_keybag)
    wrapped_volume_keybag = self.get_wrapped_keybag(volume_keybag_start_addr, volume_keybag_block_count)

    # Decrypts the wrapped volume keybag
    # THIS IS STEP 5 OF THE APFS ACCESSING ENCRYPTED OBJECTS DOCUMENTATION
    unwrapped_volume_keybag = self.decrypt_keybag(wrapped_volume_keybag, volume_keybag_start_addr, volume_uuid)

    # Parses the unwrapped volume kb_locker object derived from the wrapped keybag into structures
    parsed_volume_keybag = parse_volume_keybag(unwrapped_volume_keybag)
    print(unwrapped_volume_keybag)

~ 종략 ~

```

뿐만 아니라 checkpoint 부분에서 일부 Container Signature 누락으로 아래와 같이 apfs_reader.py 를 수정하였다.

```
Traceback (most recent call last):
  File "sin90_test.py", line 417, in <module>
    tryingdecryptor(img, img.get_size(), NXSB_location, uuid)
  File "sin90_test.py", line 311, in tryingdecryptor
    mac_info.apfs.container = ApfsContainer(img, container_size, container_start_offset)
  File "/Users/sin90/Desktop/DFC_challenge/481 - Irresponsible predecessor/mac_apt/plugins/helpers/apfs_reader.py", line 1588, in __init__
    checkpoint_blocks.append(self.read_block(base_cp_block_num + i))
  File "/Users/sin90/Desktop/DFC_challenge/481 - Irresponsible predecessor/mac_apt/plugins/helpers/apfs_reader.py", line 1668, in read_block
    block = self.apfs.readBlock(base_cp_block_num + i)
  File "/Users/sin90/Desktop/DFC_challenge/481 - Irresponsible predecessor/mac_apt/plugins/helpers/apfs.py", line 734, in __init__
    self.body = self._root.ContainerSuperblock(self._io, self, self._root)
  File "/Users/sin90/Desktop/DFC_challenge/481 - Irresponsible predecessor/mac_apt/plugins/helpers/apfs.py", line 516, in __init__
    self.magic = self._io.ensure_fixed_contents(b'NXSB')
  File "/opt/anaconda3/envs/mac_apt/lib/python3.7/site-packages/kaitaisstruct.py", line 325, in ensure_fixed_contents
    (actual, expected)
Exception: unexpected fixed contents: got b'\x00\x00\x00\x00', was waiting for b'NXSB'

checkpoint_blocks = []
for i in range(num_cp_blocks) :
    try :
        checkpoint_blocks.append(self.read_block(base_cp_block_num + i))
    except :
        pass
```

[그림 25] APFS Reader 수정 코드

수정 이후 실행 결과는 아래와 같다.

[그림 26] APFS Reader 수정 결과

Unwrapped Container Keybag의 경우 위의 그림과 같이 “syek”라는 Keybag Signature가 해당하는 offset에 존재하는 것으로 보아, Container Keybag를 decrypt하는데 필요한 Container UUID 등 정보가 이상 없음을 확인 및 복호화가 정상적으로 진행된 것을 확인하였다.

[그림 27] Container Keybag Decrypt를 위한 Container UUID 확인

또한 최종적으로 unwrapped 된 Volume Keybag 의 내용을 확인하였더니 아래와 같은 결과를 얻을 수 있었다.

[그림 28] Volume Keybag 및 Password Hint 확인

“scer”이라는 Volume Keybag를 확인하였고, Password hint로 “7 digits + 1 special character”라는 문자열을 확인할 수 있었다. 이후 최종적으로 apfs2hashcat 내에 내장되어 있는 apfs-fuse라는 도구를 이용해 아래와 같이 수정된 usb-002.dd image를 mount 시도하였다. Mount는 ubuntu 20.04 기반의 SIFT Workstation¹¹을 이용하였으며, apt-get을 이용해 fuse3를 추가 설치하였다. 아래는 Mount 시도 결과이고, 위에서 확인한 Password Hint를 재확인할 수 있었다.

```
sansforensics@siftworkstation: ~/Desktop
$ apfs-fuse -d 31 USB-002_test_GPTmodified2_4316_uuidchange_4318_voluuid_checksum.dd.dmg  ss
Found valid GPT partition table. Looking for APFS partition.
Mounting xid 110
omap: oid=1027 xid=110 flags=0 size=0 paddr=1027
omap: oid=1029 xid=110 flags=0 size=0 paddr=1029
starting LoadKeybag @ 1070
  all blocks verified
Omap Lookup: oid=402 xid=6e: oid=402 xid=6e => flags=10 size=10 paddr=10dc
Volume DataBackup is encrypted.
starting LoadKeybag @ 106f
  all blocks verified
Hint: 7 digits + 1 special character
Enter Password:
```

[그림 29] Password Hint 데이터 검증

¹¹ <https://www.sans.org/tools/sift-workstation/>

2. What is the access password for the file server? (50 points)

1번 문항을 통해 mount 가능할 정도로 USB-002.dd Image를 복구하였고, 본 문항에서는 실제로 복호화를 통해 USB-002.dd 내에 저장된 Server Password를 확인하는 것이 목표로 한다. Password Hint로 7 digits + 1 special character를 기 확인하였다. Hashcat의 경우 이러한 Encrypt APFS의 Password의 Hash를 crack하기에 적합한 도구이다. 아래와 같이 Hashcat에서 지원하는 Hash에 18300이 APFS와 관련있는 것을 확인할 수 있었다.

12900	Android FDE (Samsung DEK)	Full-Disk Encryption (FDE)
8800	Android FDE <= 4.3	Full-Disk Encryption (FDE)
18300	Apple File System (APFS)	Full-Disk Encryption (FDE)
6211	TrueCrypt RIPEMD160 + XTS 512 bit	Full-Disk Encryption (FDE)

[그림 30] Hashcat의 APFS 지원 확인

이후에는 apfs2hashcat¹²을 이용하여 Hashcat에 feeding하기 위한 Hash를 추출하였다. 추출 방법은 해당 Github Readme의 Extract the hash 부분을 참고하였다. 결과는 아래와 같다.

```
sansforensics@siftworkstation: ~/Desktop/apfs2hashcat/build
$ sudo ./apfs-dump-quick ../../USB-002_test_GPTmodified2_4316_uuidchange_4318_voluuid_checksum.dd.dmg final.txt
Info: Found valid GPT partition table on main device. Dumping first APFS partition.
starting LoadKeybag
all blocks verified
starting LoadKeybag
all blocks verified
Volume DataBackup is encrypted.
starting LoadKeybag
all blocks verified
Hint: 7 digits + 1 special character
starting LoadKeybag
all blocks verified
Dumping Keybag (recs)

Keys : 2

Key 0:

UUID      : ED26AAA3-F340-4AA1-B9C4-6294DC84ADBF
KEK Wrpd: 19A42098A291E27A515477F46B8839AD05DD6AE256A6E743012D01D8380F562C7621AF7766A551F
Iterat's: 162130
Salt      : 563F1E57988BA8D4EBFD7331685D0086

Key 1:

Hint      : 7 digits + 1 special character

Formatted hash to use with Hashcat. Check corresponding UUID.
-----
$fvde$2$16$563F1E57988BA8D4EBFD7331685D0086$162130$19A42098A291E27A515477F46B8839AD05DD6AE256A6E743012D01D8380F562C7621AF7766A551F

Password doesn't work for any key.
Wrong password!
sansforensics@siftworkstation: ~/Desktop/apfs2hashcat/build
$
```

[그림 31] Hash 추출

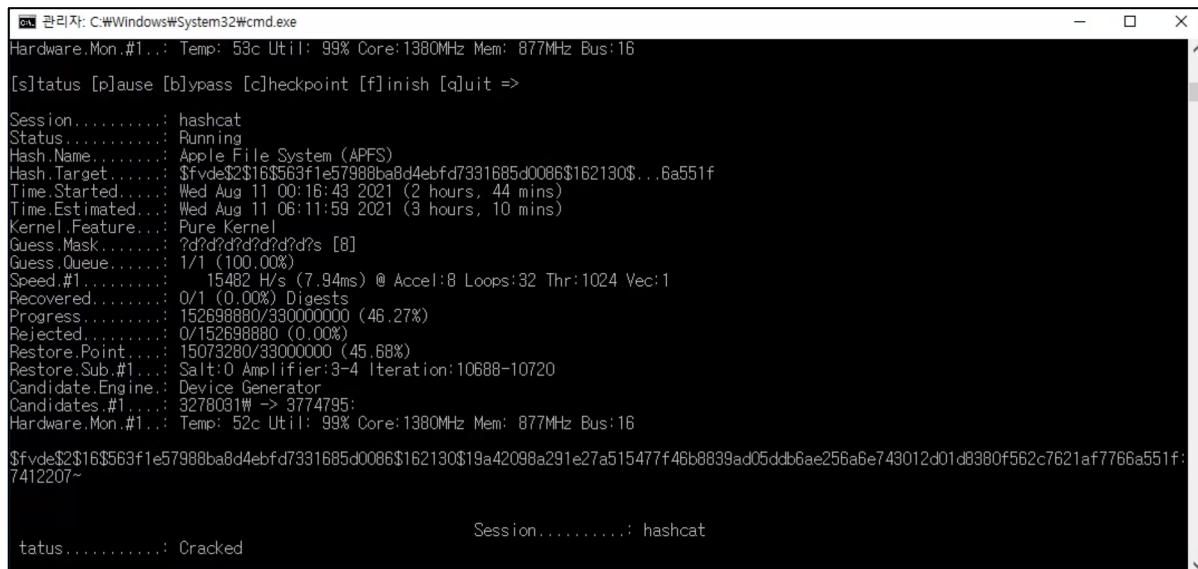
'\$fvde\$2\$16\$563F1E57988BA8D4EBFD7331685D0086\$162130\$19A42098A291E27A515477F46B8839AD05DD6AE256A6E743012D01D8380F562C7621AF7766A551F'

추출한 hash는 txt 파일로 저장하여 hashcat에 아래와 같이 적용하였다. Password hint로 7digits + 1 special character는 "?d?d?d?d?d?d?s"와 같이 마스킹을 적용하여 Bruteforce 할 수 있다.

```
hashcat -a 3 -m 18300 401hash.txt "?d?d?d?d?d?d?s"
```

¹² <https://github.com/Banaanhangwagen/apfs2hashcat>

결과는 아래와 같다. Password는 “7412207~”로 확인되었다. Dogecoin 팀 명에 걸맞는 Tesla V100을 이용하여 약 2시간 45분 소요되었다.

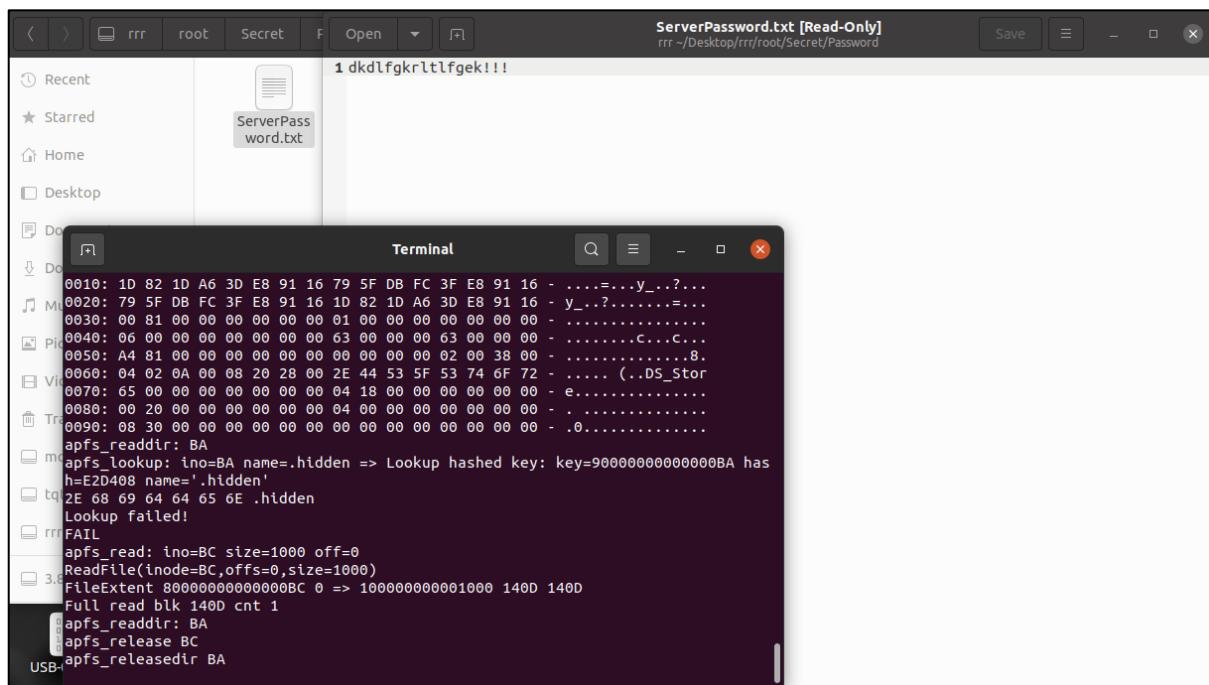


```
관리자: C:\Windows\System32\cmd.exe
Hardware.Mon.#1..: Temp: 53c Util: 99% Core:1380MHz Mem: 877MHz Bus:16
[s]tatus [p]ause [b]ypass [c]heckpoint [f]inish [q]uit =>
Session.....: hashcat
Status.....: Running
Hash.Name....: Apple File System (APFS)
Hash.Target...: $fvde$2$16$563f1e57988ba8d4ebfd7331685d0086$162130$...6a551f
Time.Started...: Wed Aug 11 00:16:43 2021 (2 hours, 44 mins)
Time.Estimated.: Wed Aug 11 06:11:59 2021 (3 hours, 10 mins)
Kernel.Feature.: Pure Kernel
Guess.Mask....: ?d?d?d?d?d?d? [8]
Guess.Queue...: 1/1 (100.00%)
Speed.#1....: 15482 H/s (7.94ms) @ Accel:8 Loops:32 Thr:1024 Vec:1
Recovered....: 0/1 (0.00%) Digests
Progress.....: 152698880/330000000 (46.27%)
Rejected....: 0/152698880 (0.00%)
Restore.Point.: 15073280/330000000 (45.68%)
Restore.Sub.#1.: Salt:0 Amplifier:3-4 Iteration:10688-10720
Candidate.Engine.: Device Generator
Candidates.#1.: 3278031# -> 3774795:
Hardware.Mon.#1..: Temp: 52c Util: 99% Core:1380MHz Mem: 877MHz Bus:16
$fvde$2$16$563f1e57988ba8d4ebfd7331685d0086$162130$19a42098a291e27a515477f46b8839ad05ddb6ae256a6e743012d01d8380f562c7621af7766a551f:
7412207~

tatus.....: Cracked
```

[그림 32] APFS Hash Crack 결과

일련의 과정을 통해 획득한 Password를 입력하고, mount한 APFS 파티션의 Secret 폴더를 확인한 결과, ServerPassword.txt 파일을 통해 아래와 같이 Server Password를 확인할 수 있었다.



[그림 33] 마운트 후 ServerPassword.txt 확인

Server Password는 “dkdlfgkrltlfgek!!!”(아일하기싫다!!!)임을 확인할 수 있었다.

- 3. Develop a tool to repair damaged parts of the given USB image file by analyzing the file system structure. (If library is required, include library or installation file) (150 points)**

위에서 설명한 과정을 기반으로, 파일 시스템 구조에 기반한 USB 이미지 복구 프로그램을 개발하였다.

■ Program Requirements

본 프로그램은 다음의 환경에서 개발 및 테스트되었으며 Dependency를 가진다.

[표 8] 프로그램 요구사항 및 테스트 환경

OS	macOS Catalina 10.15.6
Python	3.7.11 이상
Dependency Module	mac_apt ¹³ gpt_parser ¹⁴

■ Installation

본 프로그램을 정상적으로 구동시키기 위한 Dependency 모듈은 동봉되어 있지만 해당 모듈들의 작동을 위한 일련의 라이브러리가 설치 되어야한다.

본 프로그램에서 요구하는 라이브러리의 목록은 아래와 같다.

[표 9] 프로그램 필요 라이브러리 목록

라이브러리 명	버전
anytree	2.8.0
biplist	1.0.3
cffi	1.14.6
construct	2.10.67
cryptography	3.4.8
kaitaisstruct	0.9
nska-deserialize	1.3.2
numpy	1.21.2
pycparser	2.20
pyliblzfse	0.4.1
Pytsk3	20210801
six	1.16.0
xlsxWriter	3.0.1

¹³ https://github.com/n0fate/raw/blob/master/gpt_parser.py

¹⁴ https://github.com/ydkhatri/mac_apt/tree/master/plugins

프로그램을 개발하기에 앞서 라이브러리가 필요할 시, 라이브러리를 포함하거나 설치 파일을 동봉할 것을 요구하고 있으므로 해당 라이브러리를 자동으로 설치함에 필요한 requirements 를 동봉하였다.

```
(venv) dhyun@gimdonghyeon-ui-MacBookPro ~ /Documents/DFC-2021/Temp/401-sourcecode/venv/bin pip freeze > requirements.txt  
(venv) dhyun@gimdonghyeon-ui-MacBookPro ~ /Documents/DFC-2021/Temp/401-sourcecode/venv/bin cat requirements.txt  
anytree==2.8.0  
biplist==1.0.3  
cffi==1.14.6  
construct==2.10.67  
cryptography==3.4.8  
kaitaistruct==0.9  
nska-deserialize==1.3.2  
numpy==1.21.2  
pycparser==2.0.0  
pyliblzse==0.4.1  
pytsk3==20210801  
six==1.16.0  
XlsxWriter==3.0.1
```

[그림 34] PIP Freeze를 이용한 requirements 파일 생성

해당 파일을 이용한 라이브러리 설치 명령어는 아래와 같다.

[표 10] 라이브러리 설치 명령어 (pip는 사용자의 환경에 따라 버전이 상이하다.)

```
pip3 install -r requirements.txt
```

```
(venv) dhyun@gimdonghyeon-ui-MacBookPro ~ /Documents/DFC-2021/Temp/401-sourcecode/venv/bin pip install -r requirements.txt  
Requirement already satisfied: anytree==2.8.0 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 1)) (2.8.0)  
Requirement already satisfied: biplist==1.0.3 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 2)) (1.0.3)  
Requirement already satisfied: cffi==1.14.6 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 3)) (1.14.6)  
Requirement already satisfied: construct==2.10.67 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 4)) (2.10.67)  
Requirement already satisfied: cryptography==3.4.8 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 5)) (3.4.8)  
Requirement already satisfied: nska-deserialize==0.9 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 6)) (0.9)  
Requirement already satisfied: numpy==1.21.2 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 7)) (1.3.2)  
Requirement already satisfied: pycparser==2.20 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 8)) (2.20)  
Requirement already satisfied: pyliblzse==0.4.1 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 9)) (0.4.1)  
Requirement already satisfied: pytsk3==20210801 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 10)) (20210801)  
Requirement already satisfied: six==1.16.0 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 11)) (1.16.0)  
Requirement already satisfied: XlsxWriter==3.0.1 in /Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/lib/python3.9/site-packages (from -r requirements.txt (line 12)) (3.0.1)  
WARNING: You are using pip version 21.1.3; however, version 21.2.4 is available.  
You should consider upgrading via the '/Users/dhyun/Documents/DFC-2021/Temp/401-sourcecode/venv/bin/python3.9 -m pip install --upgrade pip' command.
```

[그림 35] 설치 완료 화면

정상적인 설치가 이뤄지고 난 뒤, 재설치 버튼을 누르면 라이브러리가 정상적으로 설치되어 있다는 메시지를 다음과 같이 확인할 수 있다.

■ Manual

본 프로그램은 CLI 형태로 구동되며, 인자 값으로 복구가 필요한 USB 파일 명을 요구한다.

[표 11] 프로그램 구동 명령어

```
python3 DFC_401.py USB-002.dd
```

[그림 36] 프로그램 구동 시작

[그림 37] 프로그램 구동 완료 및 복구 USB 이미지 확인

정상적으로 USB 이미지가 복구되었다면 `repaired`라는 키워드가 포함된 이미지 파일이 생성됨을 위의 그림과 같이 확인할 수 있다.

만약 복구 과정에서 시나리오 및 정의된 로직에서 벗어나는 예상치 못한 문제가 발생할 시, 아래와 같은 메시지를 출력하며 프로그램을 종료한다.

```
[96, 112]  
[32, 48]  
[144, 160]  
[64, 80]  
[160, 176]  
[128, 144]  
[16, 32]  
[112, 128]  
{0, 16}  
{80, 96}  
[176, 192]  
[*] Error occurred
```

[그림 38] USB 이미지 복구 과정에서 에러 발생 시 화면

또한 프로그램의 구조 상 1번 항목에서 서술한 데이터들을 확인해서 정상적인 것으로 판단, 복구 지점이 없는 경우, (정상적인 USB 이미지) 아래와 같은 화면을 출력함을 확인할 수 있다.

```
[96, 112]  
[32, 48]  
[144, 160]  
[64, 80]  
[160, 176]  
[128, 144]  
[16, 32]  
[112, 128]  
[0, 16]  
[80, 96]  
[176, 192]  
[*] Nothing recovered
```

[그림 39] 정상 USB 이미지 대상 구동 화면