

502 – M1 Ransom

Team Information

Team Name : DogeCoin

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

Email Address : dfc-dogecoin@naver.com

Instructions

Description. Recently, your company was affected by a ransomware attack. As a result, an important PDF file was encrypted, and the extension was changed to '*.enc'. According to the hacker's request, your company sent Bitcoin and received key file, but did not receive any decoding program.

Target	Hash (SHA256)
encrypter*	4bb2989522b48b47f9f7f2f94bcb4fb620fc6ceae3700232 2b12a94acd7414d8
confidential.p df.enc	cf238c4654ed0e5f76af209037807d2007dfa04f1f4210d 3e960ecabdbbeb742
keyfile	c163b66801e86051fd84c94b2d2b7807a60c0a2f5ceceb6 73167e54daa69390e

* We recommended that you test the malware in a safe environment.

Questions

1. Describe the process of how the ransomware encrypts the PDF files in detail. (Binary static analysis result, crypto library, encryption algorithm, key file explanation) (100 points)
2. Can the encrypted files be recovered? If you can recover the files, provide a specific solution. (Note, you should submit source code you implemented.) (200 points)

There are no restrictions on the use of programming languages (golang, python, java, c++, etc.)

The program must be executed in the following command.

```
./[program] [keyfile] [targetfile.pdf.enc]
```

And the name of the output is **[targetfile.pdf]**.

for example,

```
go run decrypter.go keyfile confidential.pdf.enc
```

```
python3 decrypter.py keyfile confidential.pdf.enc
```

```
./decrypter keyfile confidential.pdf.enc
```

As a result of executing the above command, **confidential.pdf** file should be created.

Submit your source code along with a simple compilation or execution guide.

3. Decrypt the 'confidential.pdf.enc'. (You must write the SHA-256 value of the 'confidential.pdf' file on your answer sheet.) (200 points)

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:	IDA - The Interactive Disassembler	Publisher:	Hex-Rays
Version:	7.6.210427		
URL:	https://hex-rays.com/ida-pro/		

Name:	shasum	Publisher:	Mark Shelor
Version:	5.84		
URL:	-		

Step-by-step methodology:

1. Describe the process of how the ransomware encrypts the PDF files in detail.
(Binary static analysis result, crypto library, encryption algorithm, key file explanation) (100 points)

Basic Properties	
MD5	c127e9bdb652c623a10b6b752da3b107
SHA-1	2eca429f86a42dcbe5a70ea977c2637cb5a824
SHA-256	4bb2989522b48b47f9f7f2f94bcba4fb620fc6ceae37002322b12a94acd741d8
Vhash	a2a/b5b403b100a420e4dff21a74397
SSDEEP	98304:RkOPRONEY4Z5/w6giKxm8yYWUceMkxNvJx+iiTVth3CX?IFOROS7s6n5Vz5yqXNI:bRKqBtdr13kp/zB20T+rroVCU
TLSH	T132667d42FDAC6863D2CC9270776643DC323DF6859AA192374A60EE35ADF17D59F23220
File type	Mach-O
Magic	Mach-O 64-bit executable
TrID	Mac OS X Mach-O 64-bit ARM executable (100%)
File size	6.46 MB (6775010 bytes)

[그림 1] VirusTotal을 통해 획득한 랜섬웨어 바이너리의 기본 정보

랜섬웨어로 확인되는 바이너리 “**encrypter**”를 VirusTotal에 업로드 한 결과, 해당 바이너리는 “**Mac OS X Mach-O 64-bit ARM executable**” 의 타입을 가지는 것을 확인할 수 있었다.

해당 바이너리 타입은 Apple Silicon (M1) 아키텍쳐에서 구동되는 실행 파일임을 의미한다.

Output window			Program Segmentation											
Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	T	DS
HEADER	0000000100000000	0000000100001000	R	.	X	.	L	byte	04	public	DATA	64	00	04
_text	0000000100001000	00000001002030A0	R	.	X	.	L	para	01	public	CODE	64	00	01
_symbol_stub1	00000001002030A0	00000001002033E0	R	.	X	.	L	align_32	02	public	CODE	64	00	02
_rodata	00000001002033E0	0000000100250000	R	.	X	.	L	align_32	03	public		64	00	03
_rodata	0000000100250000	00000001002D89E0	R	.	.	.	L	align_32	05	public		64	00	05
_typep(link	00000001002D89E0	00000001002DA520	R	.	.	.	L	align_32	06	public		64	00	06
_itablink	00000001002DA520	00000001002DAC90	R	.	.	.	L	align_32	07	public		64	00	07
_gopclntab	00000001002DACA0	00000001004313C0	R	.	.	.	L	align_32	08	public		64	00	08
_go_buildinfo	0000000100434000	0000000100434020	R	W	.	.	L	para	09	public		64	00	09
_nl_symbol_ptr	0000000100434020	0000000100434240	R	W	.	.	L	dword	0A	public	DATA	64	00	0A
_noprtdata	0000000100434240	0000000100450920	R	W	.	.	L	align_32	0B	public		64	00	0B
_data	0000000100450920	000000010045B620	R	W	.	.	L	align_32	0C	public	DATA	64	00	0C
_bss	000000010045B620	000000010048B190	R	W	.	.	L	align_32	0D	public	BSS	64	00	0D
_noptrbss	000000010048B190	0000000100490350	R	W	.	.	L	align_32	0E	public	BSS	64	00	0E
UNDEF	0000000100490350	0000000100490568	?	?	?	.	L	para	0F	public	XTRN	64	00	0F

[그림 2] 랜섬웨어 바이너리의 Program Segmentation 정보

바이너리가 어떠한 언어로 작성되어 컴파일 되었는지 확인하기 위해, 다양한 정보를 수집하던 중

“Program Segmentation” 정보에서 “**_gopclntab**”라는 섹션이 존재하는 것을 확인할 수 있었다.

해당 섹션은 Go Language로 작성된 바이너리에서 확인할 수 있는 섹션으로, 본 바이너리는 Go

Language로 작성되었으며 Apple Silicon (M1) 아키텍쳐에서 동작되는 랜섬웨어 임을 파악할 수 있다.

Function name	Segment	Start	Length	Locals	Arguments	R	F	L	M	S	B	T
runtime.main	_text	0000000100033490	000003F0	00000068	FFFFFFFFFFFF... R
runtime.main.func1	_text	000000010005B5F0	00000050	00000038	FFFFFFFFFFFF... R
runtime.main.func2	_text	000000010005B640	00000050	00000018	00000010 R
main.find_pdf_current_dir	_text	00000001002026D0	00000080	00000038	00000048 R	T
main.main	_text	0000000100202750	000007A0	00000198	00000188 R	T
main.find_pdf_current_dir.func1	_text	0000000100202EF0	000001A0	00000078	000000B8 R	T

[그림 3] main 함수 탐색

해당 랜섬웨어의 암호화 로직을 파악하기 위해, 주요한 로직이 존재하는 main 함수를 탐색하였다. 함수명을 기반으로 함수 목록을 탐색한 결과 “**main.main**” 함수를 확인할 수 있었고, “main” 내에 한 개의 다른 함수가 존재함을 확인할 수 있었다.

해당 함수를 기점으로 랜섬웨어를 정적 분석한 결과는 아래와 같다.

```

82     v24 = (_int64 *)&unk_10026D440;
83     runtime_newobject();
84     v58 = (void *)()>v25;
85     LODWORD(v27) = 0;
86     os_OpenFile(v23, (_int64)&key_path, 9LL, 0LL, (_int64)v27, (_int64)v28, v29, v30);
87     v3 = (char *)v30;
88     if ( v29 )
89         break;

```

[그림 4] OpenFile 함수 호출 지점

함수가 수행되고 가장 먼저 수행되는 유의미한 로직은 특정 파일을 Open 하는 함수이다.

```

func OpenFile

func OpenFile(name string, flag int, perm FileMode) (*File, error)

OpenFile is the generalized open call; most users will use Open or Create instead. It opens the named file with specified flag
(O_RDONLY etc.). If the file does not exist, and the O_CREATE flag is passed, it is created with mode perm (before umask). If
successful, methods on the returned File can be used for I/O. If there is an error, it will be of type *PathError.

▶ Example

▶ Example (Append)

```

[그림 5] Go의 OpenFile 함수 구성

Go에서 OpenFile¹을 수행할 시, 함수의 인자 값으로 3개를 요구한다. (파일 명, 모드, 권한), 해당 인자를 파악하기 위해 디컴파일 된 코드의 인자 값이 가리키는 섹션들에서 아래의 정보를 확인할 수 있었다.

```

[rodata:0000000100204496 key_path      DCB 0x2E ; .
[rodata:0000000100204497      DCB 0x2F ; /
[rodata:0000000100204498      DCB 0x6B ; k
[rodata:0000000100204499      DCB 0x65 ; e
[rodata:000000010020449A      DCB 0x79 ; y
[rodata:000000010020449B      DCB 0x66 ; f
[rodata:000000010020449C      DCB 0x69 ; i
[rodata:000000010020449D      DCB 0x6C ; l
[rodata:000000010020449E      DCB 0x65 ; e

```

[그림 6] OpenFile의 파일 명에 해당하는 인자 값 (“./keyfile”)

OpenFile의 “./keyfile”이라는 파일 명을 통해, 해당 함수는 랜섬웨어의 실행 폴더에 저장된 “keyfile”이라는 파일의 데이터를 가져오는 것을 확인할 수 있다.

O_NOFOLLOW	= 0x20000
O_NONBLOCK	= 0x800
O_RDONLY	= 0x0
O_RDWR	= 0x2
O_RSYNC	= 0x101000
O_SYNC	= 0x101000
O_TRUNC	= 0x200

[그림 7] 0x0에 해당하는 Syscall Constant

또한 Flag 와 Perm 은 0의 값을 가지는데, OpenFile Flag의 0은 Go의 Syscall²에서 “O_RDONLY”(읽기 전용)에 해당하는 것을 확인할 수 있다.

¹ <https://pkg.go.dev/os#OpenFile>

² <https://pkg.go.dev/syscall#Open>

```

110: github_com_google_tink_go_insecurecleartextkeyset_Read(
120:     v23,
121:     (_int64)&go_itab_github_com_google_tink_go_keyset_JSONReader_github_com_google_tink_go_keyset_Reader,
122:     v6,
123:     v26,
124:     (_int64)v27,
125:     (_int64)v28;
126:     github_com_google_tink_go_keyset__ptr_Handle_Public(v23, (_DWORD **)v26);
127:     v7 = (char *)v27;
128:     if ( v26 )
129:     {
130:         v8 = *(QWORD *)(v26 + 8);
131:         LABEL_46:
132:         v24 = (_int64 *)v8;
133:         v25 = (_int64)v7;
134:         v4 = runtime_gopanic();
135:         goto LABEL_47;
136:     }
137:     github_com_google_tink_go_hybrid_NewHybridEncryptWithKeyManager(v23, v25, 0LL, 0LL, v27, (_int64)v28, v29, v30);
138:     v9 = (char *)v30;
139:     if ( v29 )
140:     {
141:         v10 = *(QWORD *)(v29 + 8);
142:         LABEL_45:
143:         v24 = (_int64 *)v10;
144:         v25 = (_int64)v9;
145:         runtime_gopanic();
146:         goto LABEL_46;
147:     }

```

[그림 8] Tink의 Hybrid Encryption 객체 생성 및 공개 키 데이터 로드

이후에는 구글에서 개발한 다중언어, 크로스 플랫폼 암호화 라이브러리인 Tink³의 객체를 생성하거나 메서드를 사용하는 지점을 확인할 수 있다. 공식 사이트에서 Tink 의 Go 지원 여부를 확인할 수 있으며 본 랜섬웨어는 해당 라이브러리를 사용한 것으로 확인된다.

해당 지점을 분석한 결과, 첫 `insecurecleartextkeyset_Read`⁴ 함수에서 `keyset_JSONReader Reader`⁵와 이전에 로드 한 “`./keyfile`”의 데이터를 인자로 받는 것을 확인할 수 있다.

해당 함수과 인자 값을 라이브러리 코드 상에서 Tracking 한 결과, 해당 디컴파일 코드는 JSON 형태의 KeySet 데이터를 인자 값으로 받고, 읽어 들인 KeySet 에 대한 Reader 객체를 반환하는 코드라는 것을 확인할 수 있었다. 해당 키 파일 (KeySet 데이터) 객체는 Handle_Public 에 의해 읽어 들인 JSON KeyFile 이 공개 키 (Public) 객체로써 사용되는 것을 확인할 수 있다.

```

func NewJSONReader
func NewJSONReader(r io.Reader) *JSONReader

NewJSONReader returns new JSONReader that will read from r.

func (*JSONReader) Read
func (bkr *JSONReader) Read() (*tinkpb.Keyset, error)

Read parses a (cleartext) keyset from the underlying io.Reader.

```

[그림 9] Tink 라이브러리의 KeySet 로드 관련 함수 구성

이후에는 읽어 들인 키 파일 정보와 객체를 기반으로 “`NewHybridEncryptWithKeyManager`” 객체를 선언하여 암호화 함수를 사용할 수 있도록 한다.

³ <https://github.com/google/tink>

⁴ <https://github.com/google/tink/blob/master/go/insecurecleartextkeyset/insecurecleartextkeyset.go>

⁵ https://github.com/google/tink/blob/master/go/keyset/json_io.go

```

148 v56 = v28;
149 v46 = v27;
150 log_Printf(v23, (_int64)"Searching for pdf files that exist in the current directory...\n", 64LL, 0LL, 0LL, 0LL);
151 log_Printf(v23, (_int64)"Note: This program will not delete the original file.\n", 54LL, 0LL, 0LL, 0LL);
152 os_Getwd(v23, (_int64)v24, (void **)v25, v26, (_int64)v27);

```

[그림 10] 문장 출력 및 현재 디렉토리 정보 획득

```

func Getwd
func Getwd() (dir string, err error)

Getwd returns a rooted path name corresponding to the current directory. If the current directory can be reached via multiple paths
(due to symbolic links), Getwd may return any one of them.

```

[그림 11] Go의 Getwd 함수 구성

위에서 Tink 라이브러리 관련 지점을 지나면, 문장을 출력하고 Getwd 함수를 호출 함으로써 해당 랜섬웨어가 동작하는 디렉토리의 정보(경로 값)를 가져오는 것을 확인할 수 있다. 출력하는 문장은 현재 디렉토리에 존재하는 PDF 파일을 찾으며, 해당 프로그램은 원본 파일을 삭제하지 않는다는 내용의 문장이 출력된다.

```

166 main_find_pdf_current_dir(v23, v58, v25);
167 path_filepath_Walk(v23, (_int64)v52, (_int64)v40, (void **)(void))v25, (_int64)v27, (_int64)v28);
168 if ( v27 )
169 {
170     v24 = (_int64 *)v27[1];
171     v25 = (_int64)v28;
172     v10 = runtime_gopanic();
173     goto LABEL_49;
174 }
175 v11 = *((_QWORD *)v58 + 1);
176 if ( v11 <= 0 )
177 {
178     log_Printf(v23, (_int64)&unk_1002086B2, 24LL, 0LL, 0LL, 0LL);
179 }

```

[그림 12] main.find_pdf_current_dir, path_filepath_Walk 함수 호출

이후에는 두 개의 함수가 호출되는데 두 함수의 호출 결과에 따라 이후 로직이 분기 됨을 확인하였다. 먼저 main.find_pdf_current_dir 함수를 Tracking 한 결과는 다음과 같다.

```

67    v27 = path_len;
68    i = path_len - 1;
69    v29 = path_buf;
70    while ( (i & 0x8000000000000000LL) == 0 )
71    {
72        ch = path_buf[i];
73        if ( ch == '/' )           // /a/b/c.pdf -> c.pdf
74            break;
75        if ( ch == '.' )
76        {
77            ext_len = path_len - i;
78            dot_buf = &path_buf[i & ((i - path_len) >> 63)];
79            goto LABEL_10;
80        }
81        --i;
82    }

```

[그림 13] main_find_pdf_current_dir의 파일 정보 가공

먼저 파일 경로 값을 역으로 읽어 들이면서 “.” 문자열 (확장자) 이 들어간 문자열만 따로 분리하여 확장자만 따로 추출하게 된다.

```

83 ext_len = 0LL;
84 dot_buf = 0LL;
85 LABEL_10:
86 if ( ext_len == 4 && *(DWORD *)dot_buf == 'fdp.' )
87 {
88     v33 = v26[1];
89     v34 = *v26;
90     v35 = v26[2];
91     if ( v35 < v33 + 1 )
92     {
93         runtime_growslice(v39, qword_100272940, v34, v33, v35, v33 + 1, v40, v42, v44);
94         v36 = v41;
95         v37 = v43;
96         v38 = v46;
97         v46[2] = v45;
98         if ( runtime_writeBarrier )
99             runtime_gcWriteBarrier(v39);
100        else
101            *v46 = v41;
102        v27 = path_len;
103        v33 = v37;
104        v34 = v36;
105        v26 = v38;
106        v29 = path_buf;
107    }
108    v26[1] = v33 + 1;
109    *(QWORD *)v34 + 16 * v33 + 8) = v27;
110    if ( runtime_writeBarrier )
111        runtime_gcWriteBarrier(v39);
112    else
113        *(QWORD *)v34 + 16 * v33) = v29;

```

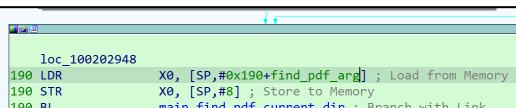
[그림 14] main.find_pdf_current_dir의 내부 조건문

[표 1] 조건문 데이터 변환 과정

1717858350 (int) → 0x6664702E (hex) → .pdf (UTF-8, Little Endian)

이전에 저장한 확장자에 대한 정보를 기반으로 확장자 길이가 4이며, “.pdf” 확장자를 가지는 파일을 찾아 경로 값을 저장하는 로직이 존재함을 확인할 수 있었다.

즉 main.find_pdf_current_dir 는 함수 명과 동일하게 주어지는 파일 경로에서 파일의 확장자를 구해서 PDF 파일을 찾아내는 함수인 것을 확인할 수 있다.

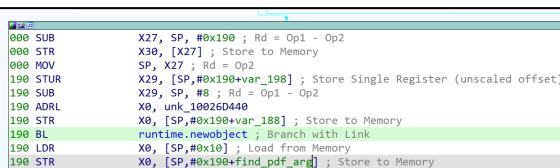


[그림 15] main.find_pdf_current_dir의 인자 Tracking

해당 함수의 인자 값을 파악하기 위해 X0 레지스터의 값을 LDR 하는 주소를 XREF로 Tracking 하였다.

Direction	Type	Address	Text
Up	w	main.main+38	STR X0, [SP,#0x190+find_pdf_arg]; Store to Memory
	r	main.main:loc_100202948	LDR X0, [SP,#0x190+find_pdf_arg]; Load from Memory
Do...	r	main.main+230	LDR X0, [SP,#0x190+find_pdf_arg]; Load from Memory
Do...	r	main.main+2B8	LDR X0, [SP,#0x190+find_pdf_arg]; Load from Memory

[그림 16] XREF를 통한 main.find_pdf_current_dir의



[그림 17] X0의 runtime.newobject 확인

그 결과, 해당 주소는 runtime.newobject 의 Return 값이 저장되는 지점을 매핑하고 있는 것을 확인할 수 있었다. 종합해보면 main.find_pdf_current_dir 함수의 인자는 찾은 PDF 파일에 대한 정보를 저장할 객체를 인자 값으로 받고 있다는 사실을 확인할 수 있다.

```
func Walk
func Walk(root string, fn WalkFunc) error

Walk walks the file tree rooted at root, calling fn for each file or directory in the tree, including root.

All errors that arise visiting files and directories are filtered by fn: see the WalkFunc documentation for details.
```

[그림 18] Go의 Walk 함수 구성

이후에는 path_filepath_Walk 라는 함수⁶가 실행되는데 이는 root 가 될 경로 값과 탐색 과정에서 수행될 함수 (WalkFunc) 를 인자 값으로 요구하는 Go 의 기본 함수다.

이에 랜섬웨어의 해당 함수는 어떤 인자 값이 실제로 사용되는지 확인을 수행하였다.

```
loc_100202948
190 LDR      X0, [SP,#0x190+find_pdf_arg] ; Load from Memory
190 STR      X0, [SP,#8] ; Store to Memory
190 BL       main.find_pdf_current_dir ; Branch with Link
190 LDR      X0, [SP,#0x10] ; Load from Memory
190 LDR      X1, [SP,#0xF0] ; Load from Memory
190 STR      X1, [SP,#8] ; Store to Memory
190 LDR      X2, [SP,#0x90] ; Load from Memory
190 STR      X2, [SP,#0x10] ; Store to Memory
190 STR      X0, [SP,#0x18] ; walk_fun
190 BL       path_filepath.Walk ; Branch with Link
190 LDR      X0, [SP,#0x190+var_170] ; Load from Memory
190 LDR      X1, [SP,#0x190+var_168] ; Load from Memory
190 LDR      X2, [SP,#0x190+var_170] ; Load from Memory
190 CBZ    X2, loc_100202E8C ; Compare and Branch on Non-Zero
```

[그림 19] path_filepath_Walk 함수 인자 Tracking

```
190 BL       os.Getwd ; Branch with Link
190 LDR      X0, [SP,#0x18] ; cwd_err
190 LDR      X1, [SP,#0x20] ; Load from Memory
190 LDR      X2, [SP,#8] ; returned_cwd_ptr
190 STR      X2, [SP,#0xF0] ; cwd_ptr
190 LDR      X3, [SP,#0x10] ; returned_cwd_len
190 STR      X3, [SP,#0x90] ; cwd_len
190 LDR      X4, [SP,#0x18] ; cwd_err
190 CBZ    X4, loc_100202948 ; Compare and Branch on Zero
```

[그림 20] X1, X2 Tracking 결과 (os.Getwd)

X0, X1, X2 레지스터의 값을 LDR 하는 주소들을 Tracking 한 결과, X0 은 main.find_pdf_current_dir 함수의 Return 값을 가리키고 있었으며 X1, X2 는 이전에 호출한 os.Getwd 의 반환 값 String (Pointer, Length) 을 확인 할 수 있었다. 즉 사용된 path_filepath_Walk 함수는 현재 디렉토리를 기준으로 main.find_pdf_current_dir 함수를 WalkFunc 로 사용하고 있는 것을 확인할 수 있다.

⁶ <https://pkg.go.dev/os#Walk>

```

177 v11 = (const char *)find_pdf_dir[11];
178 if ( (_int64)v11 <= 0 )
179 {
180     v24 = (const char *)&unk_1002086B2;
181     v25 = 24LL;
182     v26 = 0LL;
183     v27 = 0LL;
184     v28 = 0LL;
185     log_Printf();
186 }
187 else
188 {
189     v61 = 0LL;
190     v62 = 0LL;
191     v63 = 0LL;
192     v64 = 0LL;

```

[그림 21] 수집된 PDF 정보 유무 확인

이후에는 main.find_pdf_current_dir 함수에서 저장한 결과 객체의 값에 대한 비교가 수행된다. 이전에 서술하였듯이 WalkFunc로 사용되면서 수집된 PDF 정보는 특정 객체에 저장된다.

만약 PDF 파일의 정보를 저장하는 객체에서 어떤 PDF에 대한 정보도 없을 시, 즉 현재 디렉토리에 PDF 파일이 존재하지 않을 경우, 다음과 같은 문구가 출력됨을 확인할 수 있었다. (“no pdf files available.”)

```

180 v11 = *((_QWORD *)v58 + 1);
181 if ( v11 <= 0 )
182 {
183     log_Printf(v23, (_int64)&unk_1002086B2, 24LL, 0LL, 0LL, 0LL);
184 }

```

[그림 22] PDF 파일이 없을 시 출력되는 지점

._rodata:00000001002086B2 unk_1002086B2	DCB 0x6E ; n	; DATA XREF: main.main!loc_100202E447o
._rodata:00000001002086B3	DCB 0x6F ; o	
._rodata:00000001002086B4	DCL 0x20	
._rodata:00000001002086B5	DCL 0x50 ; P	
._rodata:00000001002086B6	DCL 0x44 ; D	
._rodata:00000001002086B7	DCL 0x46 ; F	
._rodata:00000001002086B8	DCL 0x20	
._rodata:00000001002086B9	DCL 0x66 ; f	
._rodata:00000001002086BA	DCL 0x69 ; i	
._rodata:00000001002086BB	DCL 0x6C ; l	
._rodata:00000001002086BC	DCL 0x65 ; e	
._rodata:00000001002086BD	DCB 0x73 ; s	

[그림 23] PDF 파일이 없을 시 출력되는 문구 데이터

```

185 else
186 {
187     v61 = 0LL;
188     v62 = 0LL;
189     v63 = 0LL;
190     v64 = 0LL;
191     runtime_convT64(v23, v11, (char *)v25);
192     v61 = &unk_100272180;
193     v62 = v25;
194     runtime_conv TString(v23, v52, (_int64)v40, v26);
195     v63 = &qword_100272940;
196     v64 = v26;
197     log_Printf(v23, (_int64)"There are %d PDF files in %s directory.\n", 40LL, (_int64)&v61, 2LL, 2LL);
198     v12 = *(_int64 **)v58;

```

[그림 24] PDF 파일이 존재할 시 출력되는 지점

그러나 현재 디렉토리에 파일이 존재한다면, 반환 받은 PDF 정보 (PDF의 개수) 와 현재 디렉토리 명을 인자로 현재 디렉토리 명과 존재하는 PDF 파일의 개수를 출력해준다.

```

198 v12 = *(_int64 **)v58;
199 if ( *(_int64 *)v58 + 1 ) > 0 )
200 {
201     v45 = *((_QWORD *)v58 + 1);
202     v13 = 0LL;
203     while ( 1 )
204     {
205         v44 = v13;
206         v55 = v13;
207         v50 = *v13;
208         v37 = v13[1];
209         LODWORD(v37) = 0;
210         os_OpenFile(v23, v50, v37, 0LL, (_int64)v27, (_int64)v28, v29, v30);
211         v57 = v28;
212         if ( v29 )
213         {
214             v59 = 0LL;
215             v60 = 0LL;
216             v59 = *_QWORD *(v29 + 8);
217             v60 = (void ***)v30;
218             v28 = (char *)log_Fatal(v23, &v59, 1LL, 1LL);
219         }
220         v25 = (void (*)())v57;
221         io_ReadAll(v23, (_int64)&go_itab__os_File_io_Reader);

```

[그림 25] PDF 파일 목록 순회 및 데이터 로드

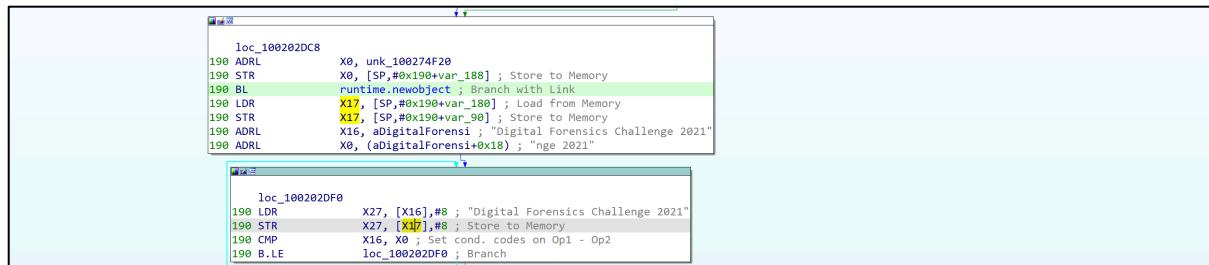
이후 수집한 PDF 정보를 순회하며, 각 PDF 파일을 열어 데이터를 읽어 들이는 로직을 수행한다.

```

248 v24 = (const char *)runtime_newobject();
249 v18 = (_QWORD *)v25;
250 v54 = v25;
251 v19 = "Digital Forensics Challenge 2021";

```

[그림 26] 정의된 문자열 (“Digital Forensics Challenge 2021”)



[그림 27] 객체 선언 및 문자열 할당

```

259 v21 = v54;
260 v14 = 0LL;
261 for ( i = 31LL; (_int64)v14 < (_int64)i; --i )
{
263     if ( v14 >= 0x20 )
264         runtime_panicIndex(v23);
265     v16 = (*(_BYTE *) (v21 + v14));
266     if ( i >= 0x20 )
267         runtime_panicIndex(v23);
268     *(_BYTE *) (v21 + v14) = *(_BYTE *) (v21 + i);
269     *(_BYTE *) (v21 + i) = v16;
270     ++v14;
271 }

```

[그림 28] 문자열 변형 코드

이후 “Digital Forensics Challenge 2021”이라는 문자열에 대해 변형하는 코드가 존재하는 것을 확인할 수 있었다. IDA 상에서 디컴파일 된 코드로는 어떠한 변형이 발생하는지 확인하기 어려워 해당 코드를 Python 으로 재구성하여 실행해보았다.

```

1 i = 31
2 j = 0
3 v86 = bytearray(b'Digital Forensics Challenge 2021')
4
5 for i in range(i, j, -1):
6     if(i < j):
7         break
8     if(j >= 0x20):
9         break
10    v71 = v86[j]
11    if(i >= 0x20):
12        break
13    v86[j] = v86[i]
14    v86[i] = v71
15    j += 1
16
17 print(v86)

```

[그림 29] Python으로 재구성한 디컴파일 코드

```
(venv) dhyun@gimdonghyeon-Ui-MacBookPro ~ /PycharmProjects/DFC-2021-502 python3 main.py
bytearray(b'1202 egnellahC scisneroF latigiD')
```

[그림 30] 재구성 코드 실행 결과

그 결과, 해당 변형 코드는 “Digital Forensics Challenge 2021” 이라는 문자열을 뒤집는 코드임을 확인할 수 있었다. (“1202 egnellahC scisneroF latigiD”)

```

272 v25 = *(int __golang **)(int, int, int, int))(v46 + 24);
273 v24 = v56;
274 v25 = (_int64)v51;
275 v26 = (_int64)v38;
276 v27 = (_int64)v39;
277 v28 = v21;
278 v29 = 32LL;
279 v30 = 32LL;
280 ((void (*)(void))v27)();

```

[그림 31] Indirect 형태의 함수 호출

이후에 **Indirect 형태의 함수 호출** (v17)이 발생하는데, 해당 함수가 어떤 객체에서 발생하였는지, 인자 값 등을 확인하기 위해 이를 Tracking 하였다.

```
loc_100202A5C
190 LDR      X1, [SP,#0x190+var_D0] ; Load from Memory
190 LDR      X2, [X1,#0x18] ; Load from Memory
190 LDR      X3, [SP,#0x110] ; Load from Memory
190 STR      X3, [SP,#8] ; Store to Memory
190 LDR      X4, [SP,#0xE8] ; Load from Memory
190 STR      X4, [SP,#0x10] ; data_ptr
190 LDR      X4, [SP,#0x80] ; Load from Memory
190 STR      X4, [SP,#0x18] ; data_len
190 LDR      X4, [SP,#0x88] ; Load from Memory
190 STR      X4, [SP,#0x20] ; data_capacity
190 STR      X0, [SP,#0x28] ; ctx_ptr
190 MOV      X0, #0x20 ; ` ` ; Rd = Op2
190 STR      X0, [SP,#0x30] ; ctx_len
190 STR      X0, [SP,#0x38] ; ctx_cap
190 BLR      X2 ; Encrypt Function
```

[그림 32] v17 호출에 대한 어셈블리 코드

V17 함수의 주소는 어셈블리 상에서 X2 레지스터에 저장되는 것을 확인할 수 있다. X2 를 확인해보면 X1 에서 0x18 사이즈만큼 더해진 주소에서 불러오며 X1 은 #0x190+var_D0 에서 불러오는 것을 확인할 수 있다. 이에 #0x190+var_D0 를 Tracking 하였다.

```
loc_1002028B0
190 STR      X3, [SP,#0x190+var_80] ; Store to Memory
190 STR      X2, [SP,#0x190+var_D0] ; Store to Memory
```

[그림 33] SP, #0x190+var_D0에 대한 Tracking

그 결과, X2의 값을 #0x190+var_D0에 저장하는 것을 확인할 수 있다.

```
loc_10020287C
199 NOP ; No Operation
199 STR X0, [SP,#0x190+var_188] ; Store to Memory
199 STP XZR, XZR, [SP,#0x190+var_188] ; Store Pair
199 BL github.com_google_tink_go_hybrid.NewHybridEncryptWithKeyManager ; Branch with Link
199 LDR X0, [SP,#0x190+var_160] ; Load from Memory
199 LDR X1, [SP,#0x190+var_158] ; Load from Memory
199 LDR X2, [SP,#0x190+var_178] ; Load from Memory
199 LDR X3, [SP,#0x190+var_168] ; Load from Memory
199 LDR X4, [SP,#0x190+var_160] ; Load from Memory
```

[그림 34] EncryptWithKeyManager 객체

X2 레지스터를 다시 Tracking하면, 이전에 선언한 Tink의 NewHybridEncryptWithKeyManager의 객체를 가리키고 있음을 확인할 수 있다.

```
• rodata:0000001002D1548 _go.itab._github.com_google_tink_go_hybrid.wrappedHybridEncrypt.github.com_google_tink.go_tink.HybridEncrypt DCQ unk_100287980  
• rodata:0000001002D1548 ; DATA XREF: github.com_google_tink_go_hybrid.NewHybridEncryptWithKeyManager+0C0  
• rodata:0000001002D1548 ; _itablink:0000001002D5AE81o  
• rodata:0000001002D1550 DCQ unk_10027F120  
• rodata:0000001002D1558 DCB 0x1F  
• rodata:0000001002D1559 DCB 0x19  
• rodata:0000001002D155A DCB 0xEA  
• rodata:0000001002D155B DCB 0x3E ; >  
• rodata:0000001002D155C DCB 0  
• rodata:0000001002D155D DCB 0  
• rodata:0000001002D155E DCB 0  
• rodata:0000001002D155F DCB 0  
• rodata:0000001002D1560 DCQ github.com_google_tink_go_hybrid. ptr wrappedHybridEncrypt .Encrypt
```

[그림 35] _ptr_wrappedHybridEncrypt_.Encrypt 함수에 대한 주소

이에 해당 객체의 주소로부터 0x18 사이즈 ($0x1002D1548 + 0x18 == 0x1002D1560$) 를 더한 결과, v17 이 **wrappedHybridEncrypt_.Encrypt** 함수⁷ 를 의미하며 인자 값을 기반으로 실제 암호화 (Hybrid Encryption) 를 수행하는 함수인 것을 파악할 수 있었다.

https://github.com/google/tink/blob/14d1865d7777a699c7871a315992709e833673d5/go/hybrid/hybrid_encrypt_factory.go

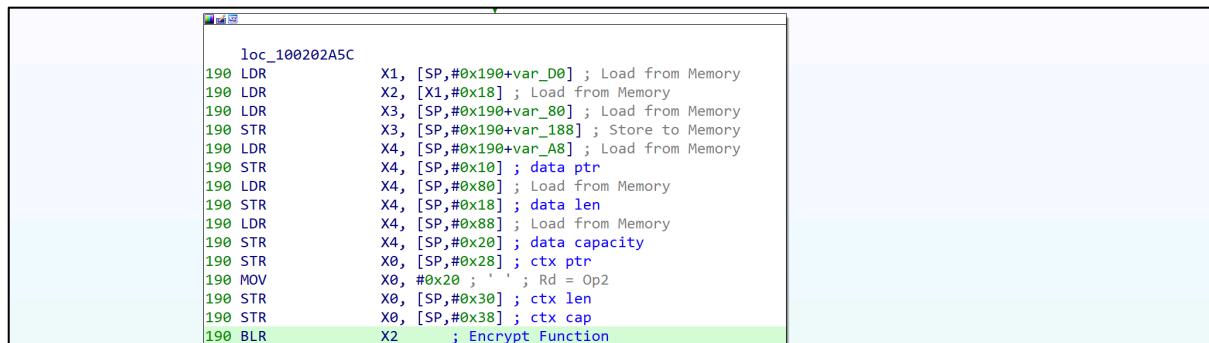
```

49
50     {
51         github_com_google_tink_go_hybrid_newEncryptPrimitiveSet();
52         result = v19;
53         a13 = &go_itab__github_com_google_tink_go_hybrid_wrappedHybridEncrypt_github_com_google_tink_HybridEncrypt;
54         a14 = v19;
55         a15 = v28;
56         a16 = v21;
57     }
      return result;

```

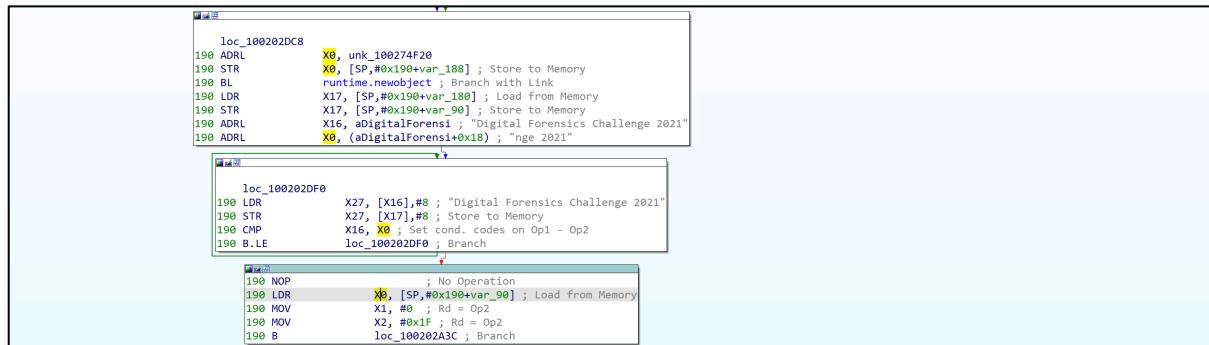
[그림 36] NewHybridEncryptWithKeyManager 내부의 반환 객체

실제로 NewHybridEncryptWithKeyManager 내부에서 반환한 객체 (HybridEncrypt) 의 주소를 가지고 있음을 확인할 수 있다.



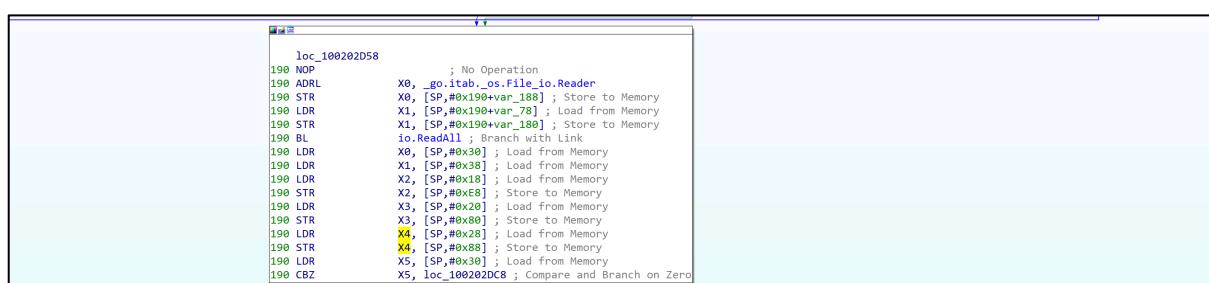
[그림 37] HybridEncrypt의 인자 값 확인 (X4, X0)

해당 함수가 암호화를 수행할 때 필요한 인자를 파악하기 위해, 함수의 인자 값이 경유 되는 것으로 확인되는 X4, X0 레지스터에 LDR 하는 주소들을 파악해 Tracking 하였다.



[그림 38] 뒤집은 문자열 (X0)

그 결과 X0 은 “Digital Forensics Challenge 2021”을 저장한 객체를 가리키고 있었으며, 해당 문자열은 후에 문자열을 뒤집는 과정에 의해 변형된 값으로 함수의 인자로 전달된다.



[그림 39] 읽어 들인 PDF 데이터 (X4)

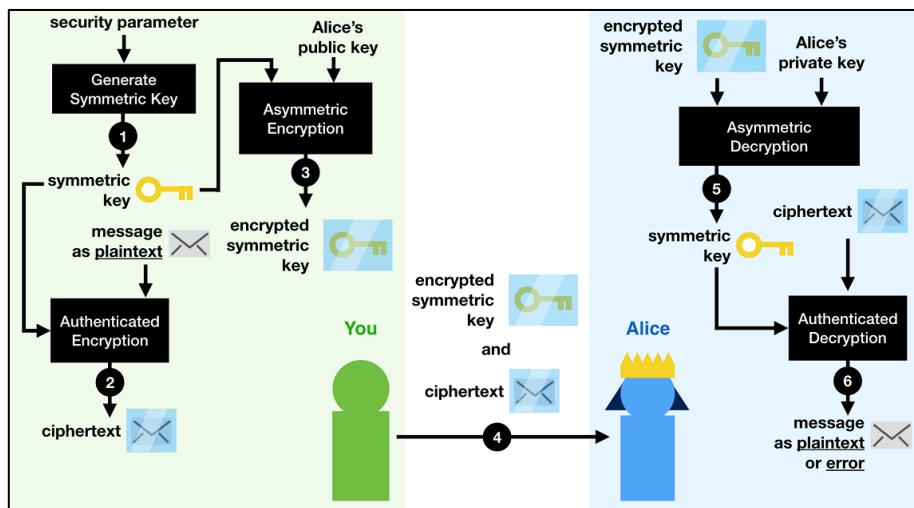
또한 X4는 이전에 읽어 들인 PDF 데이터를 가리키고 있음을 확인할 수 있었다.

즉 Hybrid Encryption 함수는 특수한 문자열과 PDF 파일 데이터를 인자로 가지는 것을 확인하였다. 다만 해당 함수가 인자를 어떠한 목적으로 활용/암호화하는지 확인하기 위해 아래의 분석 과정을 거쳤다.

Go	
Primitive	Go Implementations
AEAD	AES-GCM, AES-CTR-HMAC, KMS Envelope, CHACHA20-POLY1305, XCHACHA-POLY1305
Streaming AEAD	AES-GCM-HKDF-STREAMING, AES-CTR-HMAC-STREAMING
Deterministic AEAD	AES-SIV
MAC	HMAC-SHA2, AES-CMAC
PRF	HKDF-SHA2, HMAC-SHA2, AES-CMAC
Digital Signatures	ECDSA over NIST curves, Ed25519
Hybrid Encryption	ECIES with AEAD/DeterministicAEAD and HKDF

[그림 40] Tink Go Hybrid Encryption 알고리즘

먼저 Hybrid Encryption 이 어떠한 알고리즘에 기반하여 구현되었는지 레퍼런스를 확인해본 결과, “**ECIES with AEAD/DeterministicAEAD and HKDF**” 인 것을 확인할 수 있었다. 이에 대한 설명은 아래와 같다.



[그림 41] Hybrid Encryption 개요도

Hybrid Encryption 은 대칭 키 (Symmetric) 암호화 방식과 비대칭 키 (Asymmetric) 암호화 방식의 장점을 융합한 암호화 방식이다. 사용자가 암호화 할 메시지는 대칭 키로 암호화하고 메시지 암호화에 사용된 대칭 키는 공개 키 (Asymmetric, Public)로 암호화 하는 방식이다.

앞서 살펴보았듯이 Tink에서 구현된 Hybrid Encryption⁸은 ECIES (Elliptic Curve Integrated Encryption Scheme) Hybrid Encryption 을 기반으로 하고 있다. 해당 과정 중에서 메시지를 암호화 할에 있어 AEAD/DeterministicAEAD (Authenticated Encryption with Associated Data) 을 사용한다. 또한 사용한 대칭 키를 암호화 할 비대칭 키 암호화는 ECC 알고리즘⁹ (HKDF 키 파생 함수)을 사용하는 혼합된 방식이다. 즉 Tink에서 Hybrid Encryption 을 수행하기 위해서는 암호화에 사용할 공개 키, AEAD에 사용될 Associated 데이터, 암호화의 대상이 되는 데이터 총 3 가지가 필요하다.

바이너리 분석 결과를 검토해보면 이전에 Tink KeyManager에서 JSON으로 읽어들인 "./keyfile"이 암호화에 사용할 공개 키에 해당한다. 뒤집은 문자열 ("Digital Forensics Challenge 2021")은 Hybrid Encryption에 사용되는 AEAD의 Associated Data (Additional Data)에 해당하며 탐색할 PDF 파일 데이터가 암호화할 데이터로 볼 수 있다. 이를 인자로 Encryption을 수행하게 되는 것을 확인할 수 있다.

즉 v17 함수는 읽어들인 공개 키에 기반하여 암호화에 필요한 Associated Data 값과 PDF 파일 데이터를 인자로 암호화를 수행하는 함수이다.

```

1  {
2      "primaryKeyId": 2248856529,
3      "key": [
4          {
5              "keyData": {
6                  "typeUrl": "type.googleapis.com/google.crypto.tink.EciesAeadHkdfPrivateKey",
7                  "value": "EooBekQKBgCEAMSOhIACjB0eXBULmdvb2dsZWFwaXMuY29tL2dvb2dsZS5jcnlwG8udGluay5BZXNHY21LZXkSAhAQGAEYARogwznmNb/0"
8                  "keyMaterialType": "ASYMMETRIC_PRIVATE"
9              },
10             "status": "ENABLED",
11             "keyId": 2248856529,
12             "outputPrefixType": "TINK"
13         }
14     ]
15 }
```

[그림 42] 제공 받은 keyfile 확인

위에서 분석한 랜섬웨어 암호화 방식이 올바른지 확인하기 위해 제공 받은 키 파일을 분석해보았다. 그 결과, 해당 키 파일은 Tink 라이브러리를 통해 생성된 JSON 형태의 키로써 Type 0이 Hybrid Encryption에 사용된 "EciesAeadHkdfPrivateKey" 인 것을 확인할 수 있었다.

또한 MaterialType이 "ASYMMETRIC_PRIVATE"으로 이를 통해 해당 키 파일이 랜섬웨어에서 사용한 암호화 알고리즘에서 데이터를 복호화 하는데 쓰여지는 개인 키의 형태를 가지는 것을 확인할 수 있다. 다만 해당 키 파일이 실제 랜섬웨어 (encrypter) 가 암호화한 데이터의 복호화에 유효한 개인 키 파일인지는 추후 검증이 필요하다.

⁸ <https://developers.google.com/tink/hybrid>

⁹ <https://cryptobook.nakov.com/asymmetric-key-ciphers/ecies-public-key-encryption>

```

286     runtime_concatstring2(v23, 0LL, (_int64)v96, (_int64)v37, (_int64)&unk_100203637, 4LL, v29, [0]);
287     v48 = (Const char *)v29;
288     v36 = (QWORD *)v29;
289     LODWORD(v27) = 438;
290     os_OpenFile(v23, v29, [0], 1537LL, v27, v28, v29, [0]);

```

[그림 43] runtime_concatstring2 및 os_OpenFile 함수 호출

이후 함수에서 runtime_concatstring2 함수가 호출되며, 이에 대한 결과 값을 os_OpenFile의 인자 값으로 사용하는 것을 확인할 수 있었다. runtime_concatstring2 함수의 인자 값으로는 PDF 파일 명과 특수한 문자열이 들어가는 것이 파악되었다.

```

0000100203637 enc_ext    DCB 0x2E ; .
0000100203638          DCB 0x65 ; e
0000100203639          DCB 0x6E ; n
000010020363A          DCB 0x63 ; c

```

[그림 44] 랜섬웨어 확장자 데이터 (.enc)

특수한 문자열을 Tracking 한 결과 “.enc”라는 문자열을 확인할 수 있었다.

```

58 func concatstring2(buf *tmpBuf, a [2]string) string {
59     return concatstrings(buf, a[:])
60 }
61
62 func concatstring3(buf *tmpBuf, a [3]string) string {
63     return concatstrings(buf, a[:])
64 }
65
66 func concatstring4(buf *tmpBuf, a [4]string) string {
67     return concatstrings(buf, a[:])
68 }
69
70 func concatstring5(buf *tmpBuf, a [5]string) string {
71     return concatstrings(buf, a[:])
72 }

```

[그림 45] Runtime의 concatstring2 함수

concatstring2 함수를 분석한 결과, 두 문자열을 결합하여 반환하는 함수로써 암호화를 시도한 PDF 파일의 이름에 “.enc”라는 문자열 (확장자)를 추가한 것을 확인할 수 있다.

이후 반환 값을 os.OpenFile 함수의 인자로 사용함으로써 파일 명에 “.enc” 확장자가 추가된 새로운 파일을 생성하고자 하는 것을 확인할 수 있다.

```

310     v26 = v49;
311     if ( runtime_deferproc() )
312         break;
313     os__ptr_File_Write(v23, (_int64)v49, v53, v41, v42);
314     v43 = (const char *)v28;

```

[그림 46] File Write에 대한 함수

이후에 File Write 함수를 확인할 수 있는데, 어떠한 데이터를 Write 하는지 인자 값을 Tracking 하였다.

[그림 47] File Write 함수의 인자 Tracking (SP, #0x190+var_98)

File Write 함수의 인자 값 (SP, #0x190+var_f8) 으로 사용된 것으로 파악되는 X0 레지스터의 데이터 주소를 파악하기 위해, 확인한 결과, #0x190+var_98에서 로드한 것을 확인할 수 있었다.

190 BLR X2 ; Encrypt Function
190 LDR X0, [SP,#0x58] ; Load from Memory
190 LDR X1, [SP,#0x60] ; Load from Memory
190 LDR X2, [SP,#0x40] ; Load from Memory
190 STR X2, [SP,#0x190+var_98] ; Store to Memory

[그림 48] Encryption 함수의 반환 값

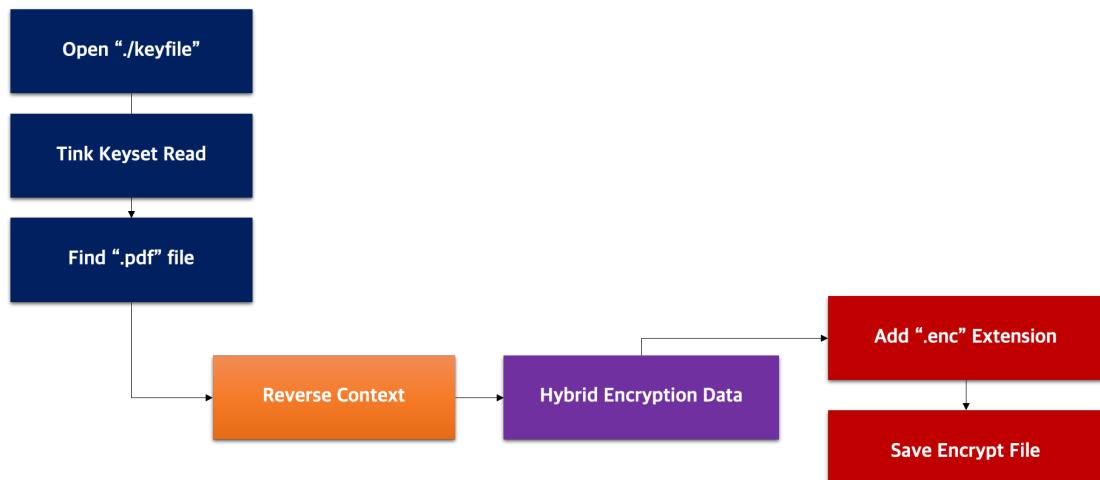
해당 주소를 Tracking 한 결과, Encryption Function (v17)의 반환 값으로 이어지는 것을 확인할 수 있었다. 즉 File Write 함수는 암호화 된 PDF 데이터를 OpenFile 로 연 파일 객체의 버퍼에 쓰고 있음을 위와 같이 파악할 수 있다.

314 runtime_convTstring(v23, v50, v37, v26);
315 v65 = &qword_100272940;
316 v66 = v26;
317 runtime_convTstring(v23, v48, v36, v26);
318 v67 = &qword_100272940;
319 v68 = v26;
320 runtime_convT64(v23, (unsigned __int64)v43, (char *)v25);
321 v69 = &unk_100272180;
322 v70 = v25;
323 log_Print(v23, ("__int64")%s -> %s : Wrote %d encrypted bytes.\n", 37LL, (__int64)&v65, 3LL, 3LL);

[그림 49] 암호화 결과에 대해 출력하는 지점

암호화 데이터를 쓰기가 완료된다면, 이전 파일 명과 암호화 이후의 파일 명, 암호화 된 파일의 크기를 포맷 스트링에 담아 출력하게 된다.

정적 분석 결과를 통해 추론한 랜섬웨어의 개요도는 아래와 같다.



[그림 50] 랜섬웨어 (encrypter) 개요도

[표 2] 랜섬웨어 분석 결과 요약

아키텍처 및 작성 언어	Apple Silicon M1, Go Language
정적 분석 결과	Tink Hybrid Encryption 을 활용한 PDF 타겟 랜섬웨어
암호화 라이브러리	Google Tink (https://github.com/google/tink)
암호화 알고리즘	ECIES with AEAD/DeterministicAEAD and HKDF
키 파일	Tink 로 생성한 개인 키 파일 (랜섬웨어 암호화 알고리즘 기반)

2. Can the encrypted files be recovered? If you can recover the files, provide a specific solution. (Note, you should submit source code you implemented.) (200 points)

(1) 복구 가능 여부

적용된 Hybrid Encryption¹⁰의 특성 상, 개인 키를 보유하고 있다면 복호화가 가능하다. 현재 비트코인을 지불함으로써 복호화에 필요한 개인 키 파일을 획득한 상황이다. 그렇기에 랜섬웨어가 사용한 암호화 라이브러리와 내부 로직을 기반으로 복호화 프로그램을 작성만 하면 **복호화가 가능할 것이다.**

Python	
Primitive	Python Implementations
AEAD	AES-GCM, AES-CTR-HMAC, AES-EAX, KMS Envelope, XCHACHA20-POLY1305
Streaming AEAD	AES-GCM-HKDF-STREAMING, AES-CTR-HMAC-STREAMING
Deterministic AEAD	AES-SIV
MAC	HMAC-SHA2, AES-CMAC
PRF	HKDF-SHA2, HMAC-SHA2, AES-CMAC
Digital Signatures	ECDSA over NIST curves, Ed25519, RSA-SSA-PKCS1, RSA-SSA-PSS
Hybrid Encryption	ECIES with AEAD/DeterministicAEAD and HKDF

[그림 51] tink Python 라이브러리의 암호화 구현 목록

항목 1에서 분석한 결과에서 랜섬웨어에서 사용된 암호화 라이브러리는 Google에서 개발한 “tink”라는 크로스 플랫폼 라이브러리로 언어 또한 다양하게 지원한다.

Go Lang에서 구현된 Hybrid Encryption은 “**ECIES with AEAD/DeterministicAEAD and HKDF**”로 Python¹¹에서도 이와 동일한 알고리즘의 Hybrid Encryption이 구현되어 있음을 확인하고 Python 기반의 복호화 프로그램을 개발하였다.

¹⁰ <https://developers.google.com/tink/hybrid>

¹¹ <https://github.com/google/tink/blob/master/docs/PYTHON-HOWTO.md>

(2) 복구 도구 매뉴얼

[표 3] 복구 도구 다운로드 링크

```
https://drive.google.com/file/d/1M_vpXzfko0TMsGYYvszMxDTtUcX2ta0P/view?usp=sharing
```

복구에 사용할 수 있는 도구는 위의 링크에서 다운로드 할 수 있다.

[표 4] 테스트 환경 및 필요 라이브러리

OS	macOS Catalina 10.15.6 (19G73)
Python	3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
Bazel	4.1.0-homebrew
Tink	1.6.1

```
(venv) dhyun@gimdonghyeon-ui-MacBookPro ~/PycharmProjects/DFC-2021-502 ➤ brew install bazel
Updating Homebrew...
==> Auto-updated Homebrew!
Updated 2 taps (homebrew/core and homebrew/cask).
==> New Formulae
alertter      bugstash      docuum       firefoxpwa    i2c-tools    licensefinder    pydocstyle    rsc_2fa     styles      wildmidi
bash-unit     doc8          fanyi        hubble       influxdb-cli  notcurses     python-launcher  sql-lint    umple
==> Updated Formulae
Updated 1172 formulae.
==> Renamed Formulae
envoy@1.17 -> envoy@1.18
wxmac -> wxwidgets
wxmac@3.0 -> wxwidgets@3.0
==> Deleted Formulae
```

[그림 52] bazel 설치 (brew)

```
(venv) dhyun@gimdonghyeon-ui-MacBookPro ~/PycharmProjects/DFC-2021-502 ➤ pip3 install tink
Collecting tink
  Using cached tink-1.6.1.tar.gz (174 kB)
Collecting absl-py
  Downloading absl_py-0.13.0-py3-none-any.whl (132 kB)
|██████████| 132 kB 1.4 MB/s
collecting protobuf
  Downloading protobuf-3.17.3-cp39-cp39-macosx_10_9_x86_64.whl (1.0 MB)
|██████████| 1.0 MB 1.9 MB/s
Collecting six
  Using cached six-1.16.0-py2.py3-none-any.whl (11 kB)
```

[그림 53] tink 라이브러리 설치 (pip3)

본 복구 도구는 위의 표에 제시된 환경에서 실행 및 테스트 되었으며, 주요 암호화 라이브러리를 컴파일 및 빌드하기 위해 “Bazel”이라는 도구를 선행으로 설치해야한다.

```
dhyun@gimdonghyeon-ui-MacBookPro ~/PycharmProjects/DFC-2021-502 ➤ python3 decrypter.py keyfile confidential.pdf.enc
[+] Get Key File : keyfile
[+] Get Encryption Data : confidential.pdf.enc
[-] Recovery Decryption Data : confidential.pdf.enc => confidential.pdf
[!] Recovery Data Header : %PDF
```

[그림 54] 복구 도구 구동 결과

설치 이후, 위의 그림과 같이 복호화에 쓰일 개인 키를 첫번째 인자, 복호화를 시도할 암호화 된 데이터를 두번째 인자로 넣어서 실행하면 복호화가 정상적으로 수행되는 것을 확인할 수 있다.

또한 복호화 여부를 확인하기 위해, 복호화 된 데이터의 4 바이트를 가져와 PDF 의 헤더를 확인할 수 있다.

(3) 복구 도구 상세 코드

여러 예제¹²를 참조하여 복구 도구를 구현하였으며, 코드에 대한 상세한 설명은 아래와 같다.

```
def __init__(self):
    self.register = hybrid.register()
    self.context = b"Digital Forensics Challenge 2021"
```

[그림 55] Context 값 설정

Hybrid Encryption 의 복호화를 위해서는 Associated Data 가 필요하므로, 정적 분석 결과를 통해 획득한 “Digital Forensics Challenge 2021” 문자열을 바이트 형태로 선언하였다.

```
def get_key_data(self, key_path):
    print("[+] Get Key File : " + key_path)
    with open(key_path, "r") as key_file:
        key_data = key_file.read()
    return key_data
```

[그림 56] Key File 로드

개인 키 파일 (JSON 형태) 을 Tink 모듈에서 사용하기 위해, 입력 받은 키 파일 경로를 기반으로 키 데이터를 읽어 들이는 함수를 선언하였다.

```
def get_enc_data(self, pdf_path):
    print("[+] Get Encryption Data : " + pdf_path)
    with open(pdf_path, "rb") as enc_file:
        enc_data = enc_file.read()
    return enc_data
```

[그림 57] 암호화 된 데이터 로드

복호화를 위해서 입력 받은 데이터 경로를 기반으로 바이트 형태로 읽어 들이는 함수를 선언하였다.

```
def reverse_context(self):
    i = 31
    j = 0
    v86 = bytearray(self.context)

    for i in range(i, j, -1):
        if(i < j):
            break
        if(j >= 0x20):
            break
```

[그림 58] Context 변환

1번 항목에서 수행한 정적 분석 결과에서 Associated Data 를 역으로 뒤집는 로직을 확인하였으며, IDA 의 Hex-rays 에서 변환한 Pseudo 코드를 기반으로 Associated Data 변환 함수를 선언하였다.

¹² <https://jryancanty.medium.com/use-googles-tink-for-asymmetric-encryption-c8957e6b5506>

```

def get_decrypt_data(self, key_path, pdf_path):
    reader = tink.JsonKeysetReader(self.get_key_data(key_path))
    kh = cleartext_keyset_handle.read(reader)
    decrypter = kh.primitive(hybrid.HybridDecrypt)
    try:
        decrypt_data = decrypter.decrypt(self.get_enc_data(pdf_path), self.reverse_context())
    except Exception as e:
        print(e)

```

[그림 59] 데이터 복호화

복호화를 수행하는 함수에서는 이전에 선언한 함수를 기반으로 개인 키, Associated 값, 암호화 된 데이터를 불러온다. 다음 Tink 의 객체에서 사용하는 **복호화 메서드의 인자 값으로 넣어 데이터 복호화를 수행한다.**

```

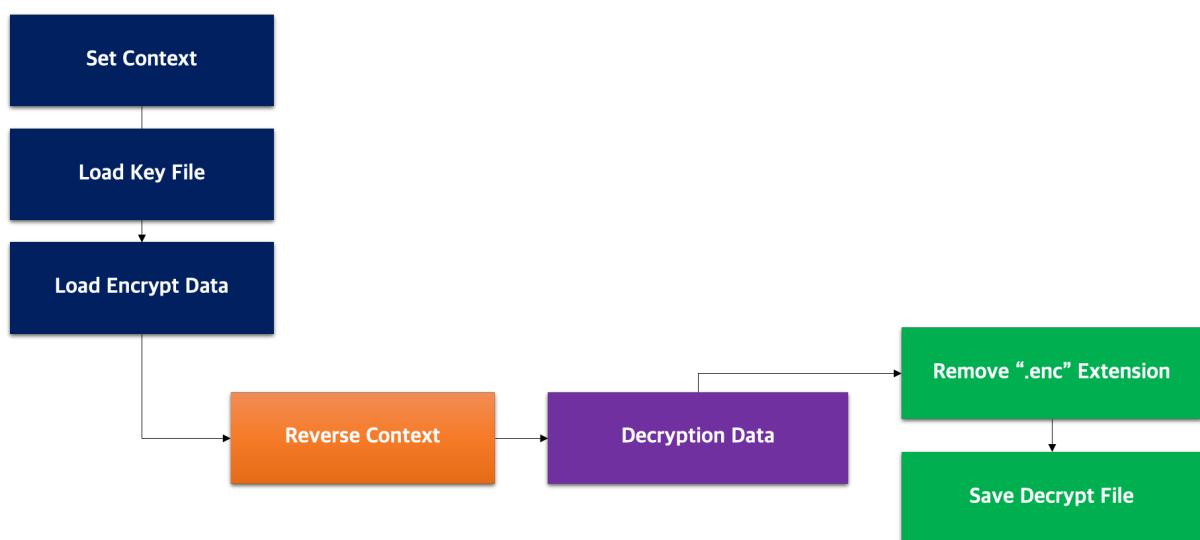
def recovery_file(self, data, pdf_path):
    original_pdf = pdf_path.replace(".enc", "")
    print("[-] Recovery Decryption Data : " + pdf_path + " => " + original_pdf)
    with open(original_pdf, "wb") as dec_file:
        dec_file.write(data)
    print("[!] Recovery Data Header : " + data[:4].decode('utf-8'))
    return

```

[그림 60] 복호화 데이터 파일로 변환

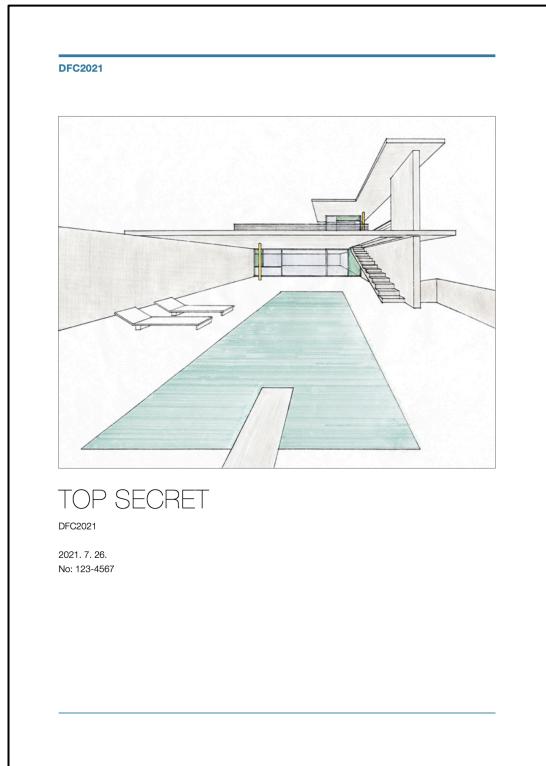
1번 항목의 정적 분석 결과에서는 암호화를 수행하고 “.enc” 확장자를 추가하는 로직이 존재하였다. 복호화 도구에서는 추가된 “.enc” 확장자를 제거하고 **복호화 데이터를 별도의 파일 형태로 저장하도록 하였다.**

위의 과정을 개요도로 표현하면 아래의 그림과 같다.



[그림 61] 복호화 도구 개요도

3. Decrypt the ‘confidential.pdf.enc’. (You must write the SHA-256 value of the ‘confidential.pdf’ file on your answer sheet.) (200 points)



[그림 62] 복호화 된 “confidential.pdf”의 표지

2 번의 과정에서 구현한 스크립트를 이용하여 첨부된 PDF 파일을 복호화 할 수 있었다. 복호화 한 PDF 파일 내부를 탐색한 결과, “TOP SECRET”라는 제목의 문서임을 확인할 수 있었다.

```
dhyun@gimdonghyeon-ui-MacBookPro ~/Desktop shasum -a 256 confidential.pdf  
33a472977c769244b6283a977e8e367dc2902cc9895e81ba363dcd3defcc0d90 confidential.pdf
```

[그림 63] “confidential.pdf”的 SHA-256 값 획득

본 항목에서는 복호화 된 “confidential.pdf”的 SHA-256 값을 기재하도록 요구하고 있다. 이에 “shasum”이라는 내장 프로그램을 통해 획득한 SHA-256 값은 아래와 같다.¹³

[표 5] “confidential.pdf”的 SHA-256 값

SHA-256	33a472977c769244b6283a977e8e367dc2902cc9895e81ba363dcd3defcc0d90
---------	--

¹³ <https://learn.akamai.com/en-us/webhelp/download-center/download-center-help/GUID-AA26AB78-5947-4A86-AA48-3C081409CCB6.html>