
REPORT



Assignment#02

Conti 랜섬웨어 분석

과목명	시큐어코딩	담당교수	이승광 교수님
학 번	32204292	전 공	모바일시스템공학과
이 름	조민혁	제 출 일	2024/11/22

목 차

1. Introduction	1
2. Requirements	2
3. Analysis	3
3-1. Static Analysis	5
3-2. Evaluation of Static Analysis	10
3-3. Paralyzation of Encrypt Function (RSA, ECRYPT)	11
3-4. Dynamic Analysis	13
4. Results	15
5. Evaluation	17
6. Conclusion	20

1. Introduction

```
All of your files are currently encrypted by CONTI strain. If you don't know who we are - just "Google it".

As you already know, all of your data has been encrypted by our software.
It cannot be recovered by any means without contacting our team directly.

DONT'T TRY TO RECOVER your data by yourselves. Any attempt to recover your data (including the usage of the additional recovery software) can damage your files. However,
if you want to try - we recommend choosing the data of the lowest value.

DONT'T TRY TO IGNORE us. We've downloaded a pack of your internal data and are ready to publish it on our news website if you do not respond.
So it will be better for both sides if you contact us as soon as possible.

DONT'T TRY TO CONTACT feds or any recovery companies.
We have our informants in these as a hostile intent and initiate the publication of whole compromised data immediatly.

To prove that we REALLY CAN get your data back - we offer you to decrypt two random files completely free of charge.

You can contact our team directly for further instructions through our website :

TOR VERSION :
(you should download and install TOR browser first https://torproject.org)

http://contirec.poc.onion/-

YOU SHOULD BE AWARE!
We will speak only with an authorized person. It can be the CEO, top management, etc.
In case you are not such a person - DON'T CONTACT US! Your decisions and action can result in serious harm to your company!
```

Fig 1. Ransom Note of Conti Ransomware

현대 사회에서 컴퓨터 시스템과 디지털 기술은 기업과 개인의 일상에서 필수적인 도구로 자리잡았다. 데이터 저장, 전송, 처리의 모든 과정이 디지털화되면서 업무의 효율성과 편리성이 극대화되었지만, 동시에 사이버 보안의 중요성도 커지고 있다. 이러한 기술의 발전과 함께 이를 악용한 사이버 범죄 역시 빠르게 증가하고 있으며, 그 중에서도 랜섬웨어는 가장 심각하고 대표적인 사이버 범죄로, 전 세계적으로 막대한 피해를 유발하고 있다.

랜섬웨어는 사용자의 파일을 암호화하거나 시스템 접근을 차단한 뒤, 이를 해제하기 위해 Fig 1 과 같이 금전적 대가를 요구하는 악성 소프트웨어다. 공격자는 암호화된 데이터의 복호화 키를 인질로 삼아 사용자로부터 비트코인과 같은 암호화폐로 대가를 요구하며, 이를 지불하지 않을 경우 데이터 유실이나 유출의 위협을 가한다. 최근 몇 년간 랜섬웨어는 단순히 데이터를 암호화하는 것을 넘어, 피해자의 데이터를 유출하고 협박하는 이중 갈취 전략으로 진화하고 있다. 이러한 랜섬웨어 공격은 개인뿐만 아니라, 기업과 국가 기관에 막대한 경제적, 법적, 신뢰도의 손실을 야기한다.

본 레포트에서는 최근 가장 악명 높은 랜섬웨어인 Conti 랜섬웨어를 분석 대상으로 선정했다. Conti 는 단순한 데이터 암호화를 넘어 고도로 조직화된 범죄 집단에 의해 설계된 랜섬웨어로, 네트워크 전파 능력, 데이터 유출 전략, 복잡한 암호화 알고리즘 등 다차원적 위협을 가한다. 특히 Conti 는 협박과 데이터 유출 전략을 결합해 피해자에게 심리적 압박을 가중하며, 복구 비용을 지불하지 않을 경우 기업의 내부 데이터를 인터넷에 공개하겠다고 위협한다.

본 레포트에서는 Conti 랜섬웨어를 대상으로 정적 분석과 동적 분석을 통해 랜섬웨어의 동작 원리와 내부 구조를 살펴본다. 레포트의 구성은 다음과 같다. 2 장에서는 Conti 랜섬웨어 분석을 위한 요구사항을 정리한다. 3 장에서는 정적 및 동적 분석을 통해 랜섬웨어의 동작과 특징을 구체적으로 살펴본다. 이후, 4 장에서는 랜섬웨어 분석에 따른 무력화 시도를 통해 결과를 평가한다. 5 장에서는 분석 결과와 그 효과를 바탕으로 종합적인 평가를 수행하며, 마지막으로 6 장에서 결론을 통해 레포트를 마무리한다.

2. Requirements

Table 1. Requirements Table

Index	Requirements
1	ELF 형식의 Conti 랜섬웨어 다운로드 받기
2	Elfinject를 이용한 바이너리 인젝션을 통해 대칭키 암호 무력화 <ul style="list-style-type: none"> a. 매개변수 분석 (평문과 암호문 버퍼) b. 암호 연산을 수행하지 않고 input 버퍼에서 output버퍼로 그대로 복사하기
3	LD_PRELOAD를 이용해 동적 라이브러리 overriding을 통해 비대칭키 암호 무력화 <ul style="list-style-type: none"> a. 비대칭키 암호 연산 무력화 후 공격된 파일에 삽입된 데이터 무엇인지 확인
4	랜섬웨어 실행 후 파일이 손상되지 않음을 보일 것 <ul style="list-style-type: none"> a. 암호화는 되지 않지만 원본 파일의 확장자 변경 b. Ransom note 등이 추가
5	랜섬웨어 실행 전/후 파일의 정보 분석을 통해 파일시스템 연산을 분석
6	랜섬웨어 실행 후 각 공격 대상 파일에 삽입된 페이로드를 분석할 것 <ul style="list-style-type: none"> a. 어떻게 획득하였는가? b. 암호화가 적용되지 않은 raw data인가? c. 크기는 무엇인가? d. 무엇인가?

Table 1 에 본 과제가 제시하는 요구사항들이 나타나 있다.

3. Analysis



Fig 2. Execution of Ransomware

3-1. Static Analysis

```

000000000004127 <main>:
4127: 55          push    rbp
4128: 48 89 e5    mov     rbp, rsp
412b: 48 83 ec 30 sub     rsp, 0x30
412f: 89 7d dc    mov     DWORD PTR [rbp-0x24], edi
4132: 48 89 75 d0 mov     QWORD PTR [rbp-0x30], rsi
4136: bf 00 10 00 00 mov     edi, 0x1000
413b: e8 20 e0 ff ff call    2160 <malloc@plt>
4140: 48 89 45 f8 mov     QWORD PTR [rbp-0x8], rax
4144: 48 8b 55 d0 mov     rdx, QWORD PTR [rbp-0x30]
4148: 8b 45 dc    mov     eax, DWORD PTR [rbp-0x24]
414b: 48 89 d6    mov     rsi, rdx
414e: 89 c7      mov     edi, eax
4150: e8 40 06 00 00 call    4795 <_Z17HandleCommandLinePPc>
4155: 8b 05 71 40 00 00 mov     eax, DWORD PTR [rip+0x4071] # 81cc <g_detached>
415b: 85 c0      test    eax, eax
415d: 0f 84 55 01 00 00 je      42b8 <main+0x191>
4163: 90         nop
4164: e8 f7 de ff ff call    2060 <fork@plt>
4169: 89 45 f4    mov     DWORD PTR [rbp-0xc], eax
416c: 83 7d f4 ff cmp     DWORD PTR [rbp-0xc], 0xffffffff
4170: 0f 85 8b 00 00 00 jne     4201 <main+0xda>
4176: 48 8d 05 9a 19 00 00 lea     rax, [rip+0x199a] # 5b17 <_ZL4note+0xad7>
417d: 48 89 c7    mov     rdi, rax
4180: e8 8b e0 ff ff call    2210 <puts@plt>
4185: 48 8d 05 84 19 00 00 lea     rax, [rip+0x1984] # 5b10 <_ZL4note+0xad0>
418c: 48 89 c6    mov     rsi, rax
418f: 48 8d 05 8f 19 00 00 lea     rax, [rip+0x198f] # 5b25 <_ZL4note+0xae5>
4196: 48 89 c7    mov     rdi, rax
4199: e8 a2 df ff ff call    2140 <fopen@plt>
419e: 48 89 45 e0 mov     QWORD PTR [rbp-0x20], rax
41a2: 48 83 7d e0 00 00 cmp     QWORD PTR [rbp-0x20], 0x0
41a7: 74 4e      je      41f7 <main+0xd0>
41a9: 8b 15 25 40 00 00 mov     edx, DWORD PTR [rip+0x4025] # 81d4 <files_encrypted>
41af: 48 8b 45 e0 mov     rax, QWORD PTR [rbp-0x20]
41b3: 48 8d 0d 77 19 00 00 lea     rcx, [rip+0x1977] # 5b31 <_ZL4note+0xaf1>
41ba: 48 89 ce    mov     rsi, rcx
41bd: 48 89 c7    mov     rdi, rax
41c0: b8 00 00 00 00 mov     eax, 0x0
:
:
:
431a: e8 3a fb ff ff call    3e59 <_Z11SearchFilesPc>
431f: e9 61 fe ff ff jmp     4185 <main+0x5e>

```

Fig 4. Disassemble of main

먼저, main 함수를 디스어셈블을 하여 관찰하였다. Fig 4 와 같이 0x431a 에서 _Z11SearchFilesPc 함수를 호출하는 것을 확인할 수 있다. 함수 이름에서 랜섬웨어 대상 파일을 탐색한다는 의미가 있다 생각하여 해당 함수를 살펴보았다.

```

000000000003e59 <_Z11SearchFilesPc>:
3e59: 55          push    rbp
3e5a: 48 89 e5    mov     rbp, rsp
3e5d: 48 83 ec 40 sub     rsp, 0x40
3e61: 48 89 7d c8 mov     QWORD PTR [rbp-0x38], rdi
3e65: bf 01 10 00 00 mov     edi, 0x1001
3e6a: e8 f1 e2 ff ff call    2160 <malloc@plt>
:
:
:
40e0: e8 54 fb ff ff call    3c39 <_Z7EncryptPv>

```

Fig 5. Disassemble of _Z11SearchFilePc

그 결과 Fig5 에서 _Z7EncryptPv 함수를 호출하는 것을 확인할 수 있었다. 함수 이름에서 Encrypt 가 있기에 해당 함수로 이동하여 분석을 진행했다.

```

00000000000003c39 <_Z7EncryptPv>:
3c39: 55                push    rbp
3c3a: 48 89 e5          mov     rbp,rsi
3c3d: 48 81 ec 30 01 00 00 sub     rsp,0x130
3c44: 48 89 bd d8 fe ff ff mov     QWORD PTR [rbp-0x128],rdi
3c4b: 48 8b 85 d8 fe ff ff mov     rax,QWORD PTR [rbp-0x128]
3c52: 48 89 85 70 ff ff ff mov     QWORD PTR [rbp-0x90],rax
3c59: 48 8b 85 70 ff ff ff mov     rax,QWORD PTR [rbp-0x90]

3cf2: e8 ed fc ff ff    call    39e4 <_Z16WriteEncryptInfoP9file_info>

3d27: e8 e5 fd ff ff    call    3b11 <_Z11EncryptFullP9file_info>

3d5c: e8 ff e3 ff ff    call    2160 <malloc@plt>

3d7c: e8 7f e3 ff ff    call    2100 <strcpy@plt>
3d81: 48 8b 45 f0        mov     rax,QWORD PTR [rbp-0x10]
3d85: 48 89 c7           mov     rdi,rax
3d88: e8 f3 e2 ff ff    call    2080 <strlen@plt>
3d8d: 48 89 c2           mov     rdx,rax
3d90: 48 8b 45 f0        mov     rax,QWORD PTR [rbp-0x10]
3d94: 48 01 d0           add     rax,rdx
3d97: c7 00 2e 63 6f 6e mov     DWORD PTR [rax],0x6e6f632e
3d9d: 66 c7 40 04 74 69 mov     WORD PTR [rax+0x4],0x6974
3da3: c6 40 06 00        mov     BYTE PTR [rax+0x6],0x0
3da7: 48 8b 85 70 ff ff ff mov     rax,QWORD PTR [rbp-0x90]
3dae: 48 89 45 e8        mov     QWORD PTR [rbp-0x18],rax
3db2: 48 8b 55 f0        mov     rdx,QWORD PTR [rbp-0x10]
3db6: 48 8b 45 e8        mov     rax,QWORD PTR [rbp-0x18]
3dba: 48 89 d6           mov     rsi,rdx
3dbd: 48 89 c7           mov     rdi,rax
3dc0: e8 eb e2 ff ff    call    20b0 <rename@plt>

```

Fig 6. Disassemble of _Z7EncryptPv

Fig 6과 같이 해당 함수는 분석에 있어서 중요한 정보를 제공하였다. 하지만 암호 연산과 관련된 로직은 찾을 수 없었다. 따라서 _Z16WriteEncryptInfoP9file_info 함수와 _Z11EncryptFullP9file_info 함수를 호출하기에 해당 함수를 분석해보았다.


```

0000000000039e4: <_Z16WriteEncryptInfoP9file_info>:
39e4: 55                push    rbp
39e5: 48 89 e5          mov     rbp, rsp
39e8: 48 83 ec 50       sub     rsp, 0x50
39ec: 48 89 7d b8       mov     QWORD PTR [rbp-0x48], rdi
39f0: 89 75 b4          mov     DWORD PTR [rbp-0x4c], esi
39f3: 89 55 b0          mov     DWORD PTR [rbp-0x50], edx
39f6: 48 8b 05 fb 47 00 00 mov     rax, QWORD PTR [rip+0x47fb] # 81f8 <g_publickey>
39fd: 48 89 c7          mov     rdi, rax
3a00: e8 9b e6 ff ff   call   20a0 <RSA_size@plt>
3a05: 48 98             cdqe
3a07: 48 89 c7          mov     rdi, rax
3a0a: e8 51 e7 ff ff   call   2160 <malloc@plt>
3a0f: 48 89 45 ff       mov     QWORD PTR [rbp-0x10], rax
3a13: 48 8b 45 b8       mov     rax, QWORD PTR [rbp-0x48]
3a17: 48 89 c7          mov     rdi, rax
3a1a: e8 d0 fe ff ff   call   38ef <_Z6GenKeyP9file_info>

3aad: 48 8b 0d 44 47 00 00 mov     rcx, QWORD PTR [rip+0x4744] # 81f8 <g_publickey>
3ab4: 48 8b 55 f0       mov     rdx, QWORD PTR [rbp-0x10]
3ab8: 48 8d 45 c0       lea     rax, [rbp-0x40]
3abc: 41 b8 04 00 00 00 mov     r8d, 0x4
3ac2: 48 89 c6          mov     rsi, rax
3ac5: bf 32 00 00 00   mov     edi, 0x32
3aca: e8 f1 e5 ff ff   call   20c0 <RSA_public_encrypt@plt>

3b04: e8 95 fe ff ff   call   399e <_Z13WriteFullDataP9file_infoPhi>

```

Fig 7. Disassemble of _Z16WriteEncryptInfoP9file_info

Fig 7 를 분석해본 결과 해당 함수는 3 개의 매개 변수를 전달받고 있다. 또한 RSA_size 와 RSA_public_encrypt 함수를 통해 비대칭키 암호 함수는 RSA 를 사용하는 것을 알 수 있었다. RSA_public_encrypt 함수는 매개변수로 [암호화 길이, 원본 데이터 버퍼, 암호화 데이터 버퍼, 키, padding 방식]으로 이루어져 있다. 여기서 원본 데이터 버퍼와 암호화 데이터 버퍼는 각각 rbp-0x40, rbp-0x10 에서 가져오는 것을 알 수 있다. 그리고, RSA 암호화를 진행하기 전 _Z6GenKeyP9file_info 함수를 호출한 후 RSA 암호화를 진행하고, _Z13WriteFullDataP9file_infoPhi 함수를 호출하는 것을 확인할 수 있다.

```

00000000000038ef <_Z6GenKeyP9file_info>:
38ef: 55                push    rbp
38f0: 48 89 e5          mov     rbp, rsp
38f3: 48 83 ec 10       sub     rsp, 0x10
38f7: 48 89 7d f8       mov     QWORD PTR [rbp-0x8], rdi
38fb: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
38ff: 48 83 c0 58       add     rax, 0x58
3903: be 20 00 00 00    mov     esi, 0x20
3908: 48 89 c7          mov     rdi, rax
390b: e8 b0 e8 ff ff    call   21c0 <RAND_bytes@plt>

3915: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
3919: 48 83 c0 50       add     rax, 0x50
391d: be 08 00 00 00    mov     esi, 0x8
3922: 48 89 c7          mov     rdi, rax
3925: e8 96 e8 ff ff    call   21c0 <RAND_bytes@plt>

395a: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
395e: 48 8d 70 58       lea     rsi, [rax+0x58]
3962: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
3966: 48 83 c0 10       add     rax, 0x10
396a: b9 40 00 00 00    mov     ecx, 0x40
396f: ba 00 01 00 00    mov     edx, 0x100
3974: 48 89 c7          mov     rdi, rax
3977: e8 24 ea ff ff    call   23a0 <ECRYPT_keysetup>

397c: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
3980: 48 8d 50 50       lea     rdx, [rax+0x50]
3984: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
3988: 48 83 c0 10       add     rax, 0x10
398c: 48 89 d6          mov     rsi, rdx
398f: 48 89 c7          mov     rdi, rax
3992: e8 df ed ff ff    call   2776 <ECRYPT_ivsetup>

```

Fig 8. Disassemble of _Z6GenKeyP9file_info

Fig 8은 _Z6GenKeyP9file_info 함수의 디스어셈블 결과를 보여준다. 먼저, RAND_bytes 함수를 두 번 호출하여 각각 ECRYPT_keysetup, ECRYPT_ivsetup 함수 호출을 통해 해당 난수들을 키와 초기화 벡터 값으로 사용하는 것을 알 수 있다. 각각의 크기는 0x20 (32 Bytes), 0x8 (8 Bytes)으로 총 40 Byte의 난수를 각각 생성한다. 난수는 rbp+0x50, rbp+0x48에 저장되는데, 이는 호출한 함수의 rbp의 변수에 대응한다. 이후 ret을 통해 호출한 함수로 돌아가서 RSA 암호화 함수를 호출한다. 이를 통해 RSA 함수에 전달되는 원본 데이터는 32 Bytes의 키와 8 Bytes의 초기화 벡터 값을 의심할 수 있다.

```

00000000000399e <_Z13WriteFullDataP9file_infoPhi>:
399e: 55                push    rbp
399f: 48 89 e5          mov     rbp, rsp
39a2: 48 83 ec 20       sub     rsp, 0x20
39a6: 48 89 7d f8       mov     QWORD PTR [rbp-0x8], rdi
39aa: 48 89 75 f0       mov     QWORD PTR [rbp-0x10], rsi
39ae: 89 55 ec          mov     DWORD PTR [rbp-0x14], edx
39b1: 8b 45 ec          mov     eax, DWORD PTR [rbp-0x14]
39b4: 48 63 d0          movsxd  rdx, eax
39b7: 48 8b 45 f8       mov     rax, QWORD PTR [rbp-0x8]
39bb: 8b 40 08          mov     eax, DWORD PTR [rax+0x8]
39be: 48 8b 4d f0       mov     rcx, QWORD PTR [rbp-0x10]
39c2: 48 89 ce          mov     rsi, rcx
39c5: 89 c7            mov     edi, eax
39c7: e8 a4 e6 ff ff   call    2070 <write@plt>
39cc: 48 85 c0          test    rax, rax
39cf: 0f 94 c0          sete    al
39d2: 84 c0            test    al, al
39d4: 74 07            je      39dd <_Z13WriteFullDataP9file_infoPhi+0x3f>
39d6: b8 00 00 00 00   mov     eax, 0x0
39db: eb 05            jmp     39e2 <_Z13WriteFullDataP9file_infoPhi+0x44>
39dd: b8 01 00 00 00   mov     eax, 0x1
39e2: c9              leave   eax
39e3: c3              ret

```

Fig 9. Disassemble of _Z13WriteFullDataP9file_infoPhi

Fig 9은 _Z13WriteFullDataP9file_infoPhi 함수의 디스어셈블 결과를 보여준다. write() 함수를 호출하는 것을 확인할 수 있으며, 해당 함수는 [파일 디스크립터, 저장할 데이터 버퍼, 바이트 수]를 매개변수로 받는다. rbp-0x10에 저장된 값을 write 하는 것을 통해 해당 버퍼의 내용이 RSA 암호화의 결과라는 것을 생각할 수 있다.

```

000000000003b11 <_Z11EncryptFullP9file_info>:
3b11: 55                push    rbp
3b12: 48 89 e5          mov     rbp, rsp
3b15: 48 83 ec 30       sub     rsp, 0x30
3b19: 48 89 7d d8       mov     QWORD PTR [rbp-0x28], rdi
3b1d: c7 45 fc 00 00 00 mov     DWORD PTR [rbp-0x4], 0x0
3b24: c7 45 f8 00 00 00 mov     DWORD PTR [rbp-0x8], 0x0
3bbc: e8 72 ec ff ff   call    2833 <ECRYPT_encrypt_bytes>

```

Fig 10. Disassemble of _Z11EncryptFullP9file_info

Fig 10은 _Z11EncryptFullP9file_info 함수의 디스어셈블 결과를 보여준다. 해당 함수는 _Z7로부터 받은 데이터를 ECRYPT_encrypt_bytes를 통해 암호화를 진행하는 것을 알 수 있다. 이를 통해 해당 랜섬웨어는 ECRYPT_encrypt_bytes를 통해 대칭키 암호 함수를 사용하는 것을 알 수 있다.

3-2. Evaluation of Static Analysis

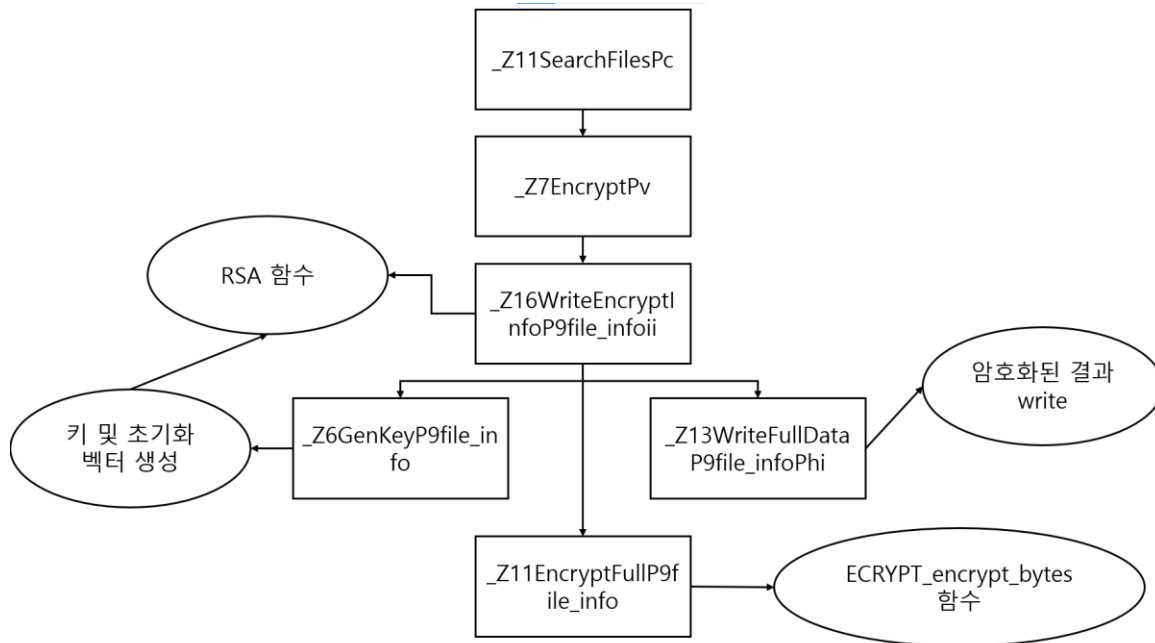


Fig 11. Process of Ransomware

3-1절에서 objdump를 활용해 정적 분석을 하였다. 이를 통해 알 수 있는 함수의 호출 순서가 Fig 11에 나타나있다. 먼저 main 함수에서 _Z11SearchFilesPc 함수를 호출한다. 해당 함수에서 _Z7EncryptPv를 호출하고, _Z16WriteEncryptInfoP9file_infoii 함수를 호출한다. 해당 함수에서 RSA 함수를 호출하는데, 매개 변수로 _Z6GenKeyP9file_info 함수에서 생성한 키 및 초기화 벡터의 값을 가져와 RSA 함수의 원본 데이터로 전달한다. 이후 _Z13WriteFullDataP9file_infoPhi 함수에서 암호화된 결과를 write하고, _Z11EncryptFullP9file_info 함수에서 ECRYPT_encrypt_bytes 함수를 호출함으로써 RSA 함수를 통해 암호화된 키 및 초기화 벡터의 값을 ECRYPT_encrypt_bytes 함수를 통해 암호화를 진행한다.

3-3. Paralyzation of Encrypt Function (RSA, ECRYPT)

본 절에서는 암호 함수를 무력화하고 출력되는 결과를 관찰한다.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
#include <string.h>
#include <openssl/rsa.h>

int RSA_public_encrypt(int flen, const unsigned char *from,
                      unsigned char *to, RSA *rsa, int padding) {

    printf("Intercepted RSA_public_encrypt!\n");
    printf("\nFlen: %d\n", flen);
    printf("\nRSA Size: %d\n", RSA_size(rsa));

    printf("\nOriginal Data Size: %d\n", strlen(from));

    printf("\nInput Data: ");

    for (int i = 0; i < strlen(from); i++) {
        printf("%02x", from[i]);
    }

    printf("\n");

    memcpy(to, 0, 512);
    memcpy(to, from, strlen(from));

    printf("\nCopy Data: ");
    for(int i = 0; i < strlen(from); i++){
        printf("%02x", to[i]);
    }
    printf("\n");
    printf("=====\\n");

    return RSA_size(rsa);
}
```

Fig 12. LD_PRELOAD of RSA_public_encrypt

본 레포트에서는 LD_PRELOAD를 통해 동적 라이브러리 함수에 대한 오버라이딩을 진행하여 해당 암호화 함수를 무력화 하였다. Fig 12와 같이 오버라이딩을 통해 원본 데이터를 암호화 버퍼에 그대로 저장하는 로직을 통해 암호화가 이루어지지 않도록 하였다.

```

BITS 64

SECTION .text
global main

main:
    ret                ; return

```

Fig 13. elfinject section of ECRYPT_encrypt_bytes - 1

3ba6:	8b 4d f4	mov	ecx,DWORD PTR [rbp-0xc]
3ba9:	48 8b 45 d8	mov	rax,QWORD PTR [rbp-0x28]
3bad:	48 8d 78 10	lea	rdi,[rax+0x10]
3bb1:	48 8b 55 e8	mov	rdx,QWORD PTR [rbp-0x18]
3bb5:	48 8b 45 e8	mov	rax,QWORD PTR [rbp-0x18]
3bb9:	48 89 c6	mov	rsi,rax
3bbc:	e8 72 ec ff ff	call	2833 <ECRYPT_encrypt_bytes>

Fig 14. ECRYPT_encrypt_bytes Call

Fig 13과 같이 단순히 return만 하도록 섹션을 제작하여 ECRYPT_encrypt_bytes 함수가 호출될 때 해당 섹션이 호출되어 아무 동작도 하지 않도록 하였다. 왜냐하면 Fig 14와 같이 ECRYPT_encrypt_bytes 함수를 호출할 때 평문 버퍼와 암호문 버퍼가 rbp-0x18로 같은 공간을 사용한다. 이에 따라 해당 섹션이 아무 것도 안하게 하여 평문이 암호화 되지 않고 평문 그대로 유효하게 존재하게 하였다.

```

BITS 64

SECTION .text
global main

main:
    push rsi
    push rdi

    lea rdi, [rdx]

    rep movsb

    pop rdi
    pop rsi
    ret                ; return

```

Fig 15. .elfinject section of ECRYPT_encrypt_bytes - 2

만약 평문 버퍼와 암호문 버퍼가 구분되어 있다면 복사를 담당하는 어셈블리어인 'rep movsb'를 호출한다. 이를 호출하기 전에 rdi 레지스터에 rdx의 버퍼 주소를 전달한다. 이를 통해 rsi 레지스터에서 rdi 레지스터로 복사가 가능하다.

```
minhyuk@minhyuk-virtual-machine:~/Assignment2/conti$ LD_PRELOAD=$(pwd)/RSA_public_encrypt.so ./conti_after --path ../all

-----
Starting encryption - CONTI POC
-----

-----
Getting into ../all
-----

Encrypting file: ../all/sample1.txt
Intercepted RSA_public_encrypt!

Flen: 50

RSA Size: 512

Original Data Size: 41

Input Data: 018e6fb698395c23f5b0965a17a411ae80ce48748097d9360beab1eb6aee1fd24693dab87bab58e264
Copy Data: 018e6fb698395c23f5b0965a17a411ae80ce48748097d9360beab1eb6aee1fd24693dab87bab58e264
```

Fig 16. Execution after Paralyzation

Fig 16에 elfinject와 LD_PRELOAD를 통한 암호화 함수 무력화를 통한 출력 결과 중 일부가 나타나있다. 해당 결과를 원본 데이터가 암호화되지 않고 output 버퍼에 그대로 복사된다는 점이다. 또한 RSA_Size가 512로 나타난 것을 통해 기존 RSA 함수는 RSA-4096를 통한 암호화를 진행했다는 것을 알 수 있다. 추가로 해당 데이터의 크기는 41바이트로 나타나있다. 정적 분석을 통해 본 레포트에서는 해당 페이로드가 32 바이트 키 + 8 바이트로 덧붙여지는 것으로 예상하였다. 그러나 41바이트의 페이로드로 덧붙여지는 것을 확인할 수 있다. 또한 페이로드의 구성 요소를 예상하였기에 검증하는 과정이 필요하다. 이를 통해 gdb를 통한 동적 분석을 수행하였다. 이는 3-4 절에서 자세히 서술한다.

3-4. Dynamic Analysis

본 절에서는 gdb를 통한 동적 분석을 수행한다. 자세하게는 정적 분석을 통한 예상 부분들을 검증하는 것을 목표로 동적 분석을 수행한다.


```
(gdb) disas
Dump of assembler code for function _Z6GenKeyP9file_info:
0x0000555555578ef<+0>:  push    rbp
0x0000555555578f0<+1>:  mov     rbp, rsp
=> 0x0000555555578f3<+4>:  sub     rsp, 0x10
0x0000555555578f7<+8>:  mov     QWORD PTR [rbp-0x8], rdi
0x0000555555578fb<+12>: mov     rax, QWORD PTR [rbp-0x8]
0x0000555555578ff<+16>: add     rax, 0x58
0x000055555557903<+20>: mov     esi, 0x20
0x000055555557908<+25>: mov     rdi, rax
0x00005555555790b<+28>: call    0x555555561c0 <RAND_bytes@plt>
0x000055555557910<+33>: cmp     eax, 0xffffffff
0x000055555557913<+36>: je      0x5555555792f <_Z6GenKeyP9file_info+64>
0x000055555557915<+38>: mov     rax, QWORD PTR [rbp-0x8]
0x000055555557919<+42>: add     rax, 0x50
0x00005555555791d<+46>: mov     esi, 0x8
0x000055555557922<+51>: mov     rdi, rax
0x000055555557925<+54>: call    0x555555561c0 <RAND_bytes@plt>

0x00005555555795a<+107>: mov     rax, QWORD PTR [rbp-0x8]
0x00005555555795e<+111>: lea     rsi, [rax+0x58]
0x000055555557962<+115>: mov     rax, QWORD PTR [rbp-0x8]
0x000055555557966<+119>: add     rax, 0x10
0x00005555555796a<+123>: mov     ecx, 0x40
0x00005555555796f<+128>: mov     edx, 0x100
0x000055555557974<+133>: mov     rdi, rax
0x000055555557977<+136>: call    0x555555563a0 <ECRYPT_keysetup>
0x00005555555797c<+141>: mov     rax, QWORD PTR [rbp-0x8]
0x000055555557980<+145>: lea     rdx, [rax+0x50]
0x000055555557984<+149>: mov     rax, QWORD PTR [rbp-0x8]
Type <RET> for more, q to quit, c to continue without paging--
0x000055555557988<+153>: add     rax, 0x10
0x00005555555798c<+157>: mov     rsi, rdx
0x00005555555798f<+160>: mov     rdi, rax
0x000055555557992<+163>: call    0x55555556770 <ECRYPT_ivsetup>
0x000055555557997<+168>: mov     eax, 0x1
0x00005555555799c<+173>: leave
0x00005555555799d<+174>: ret
```

Fig 17. gdb of _Z6GenKeyP9file_info

Fig 17에 _Z6GenKeyP9file_info 함수의 gdb 결과가 나타나있다. ECRYPTY_keysetup, ECRYPT_ivsetup이 호출되기 전에 breakpoint를 설정하여 register 값을 확인하여 key와 iv 값을 먼저 확인하도록 한다.

```
(gdb) x/32x 0x7fffffffdd48
0x7fffffffdd48: 0x1ca928a1 0xaa46a036 0xae291595 0x4e7700cc
0x7fffffffdd58: 0x23ff4154 0x79cff548 0x8f53afc3 0xa1502f48

(gdb) x/32x 0x7fffffffdd40
0x7fffffffdd40: 0xd4487675 0x9fced97a
```

Fig 18. Value of Key & IV

Fig 18을 통해 난수를 통해 생성된 Key와 IV 값을 확인할 수 있다. 이제 RSA 암호화가 수행될 때 전달되는 원본 데이터 버퍼의 값을 확인하여 덧붙여지는 값이 Key와 IV임을 확인한다.

(gdb) x/10x 0x7fffffffdc00			
0x7fffffffdc00: 0x1ca928a1	0xaa46a036	0xae291595	0x4e7700cc
0x7fffffffdc10: 0x23ff4154	0x79cff548	0x8f53afc3	0xa1502f48
0x7fffffffdc20: 0xd4487675	0x9fced97a		

Fig 19. Value of RSA's Original Data

Fig 19을 통해 Fig 18과 값이 일치하기에 RSA 함수가 호출될 때 전달되는 매개변수 값이 32바이트의 key와 8바이트의 초기화 벡터 값을 알 수 있다. 추가로 RSA 함수 호출 후 write 함수가 호출되기에 덧붙여지는 페이로드 값은 32 바이트의 key와 8 바이트의 초기화 벡터 값을 알 수 있다. 그러나 덧붙여지는 데이터는 41바이트였다. 이에 따라 1바이트가 무엇인지 확인하는 과정이 필요하다. 이는 5장에서 설명한다.

4. Results

본 장에서는 무력화된 Conti 랜섬웨어를 실행하여 실행 결과를 확인한다.

Encrypting file: ../all/sample1.txt Intercepted RSA_public_encrypt!	Encrypting file: ../all/sampleptx.ptx Intercepted RSA_public_encrypt!
Flen: 50	Flen: 50
RSA Size: 512	RSA Size: 512
Original Data Size: 41	Original Data Size: 41
Input Data: 5a0c2c139e4bd7f68150bcad7faf402ae7316b9b28bf0127c81b7d9a4985a8596b144ce385361ca064	Input Data: b7bb42d786eb02e2e70e28e40b4097c1c83cef38d8efbaa9198c886d56a9e4e8754ce5e64f90053a64
Copy Data: 5a0c2c139e4bd7f68150bcad7faf402ae7316b9b28bf0127c81b7d9a4985a8596b144ce385361ca064	Copy Data: b7bb42d786eb02e2e70e28e40b4097c1c83cef38d8efbaa9198c886d56a9e4e8754ce5e64f90053a64
Encrypting file: ../all/sample3.xlsx Intercepted RSA_public_encrypt!	Encrypting file: ../all/maps.png Intercepted RSA_public_encrypt!
Flen: 50	Flen: 50
RSA Size: 512	RSA Size: 512
Original Data Size: 41	Original Data Size: 41
Input Data: 70c5cb4580cd20a01f0188e19114a8816fc695c2893edac9832ce156cde8dd0629940dd602240ad964	Input Data: cd01e65022e622cc9b6786a1d5447735746067af1afe90062552f2846c174257328c26c4b3882d0f64
Copy Data: 70c5cb4580cd20a01f0188e19114a8816fc695c2893edac9832ce156cde8dd0629940dd602240ad964	Copy Data: cd01e65022e622cc9b6786a1d5447735746067af1afe90062552f2846c174257328c26c4b3882d0f64
Encrypting file: ../all/flags.png Intercepted RSA_public_encrypt!	Encrypting file: ../all/dummy.pdf Intercepted RSA_public_encrypt!
Flen: 50	Flen: 50
RSA Size: 512	RSA Size: 512
Original Data Size: 41	Original Data Size: 41
Input Data: 452f206025e2e3b7db04212dc2da299be31540d661b437382db91adea54d7fcb84199b72678bd0d364	Input Data: 27516bc59aca8d10dee144b8097358b0fcf260f0785cf2dc9fc73f310a06141523c6e48b7ed2061b64
Copy Data: 452f206025e2e3b7db04212dc2da299be31540d661b437382db91adea54d7fcb84199b72678bd0d364	Copy Data: 27516bc59aca8d10dee144b8097358b0fcf260f0785cf2dc9fc73f310a06141523c6e48b7ed2061b64

Fig 20. Result of Paralyzed Conti

Fig 20을 통해 모든 파일이 동일하게 41 바이트로 복사되는 것을 확인할 수 있다. 또한 언급했듯이 덧붙여지는 페이로드는 [32 바이트 key + 8 바이트 iv + 1 바이트 ?] 로 구성 되어있다. Fig 18을 관찰하면 모두 동일하게 0x64가 덧붙여지는 것을 확인할 수 있다. 이를 통해 해당 바이트는 1 바이트의 시그니처 또는 구분자인 것을 알 수 있다.

```

00000180: 613f 0a0a 5175 616d 7175 616d 2069 6420 a?..Quamquam id
00000190: 7175 6964 656d 206c 6963 6562 6974 2069 quidem licebit i
000001a0: 6973 2065 7869 7374 696d 6172 652c 2071 is existimare, q
000001b0: 7569 206c 6567 6572 696e 742e 2053 756d ui legerint. Sum
000001c0: 6d75 6d20 6120 766f 6269 7320 626f 6e75 mum a vobis bonu
000001d0: 6d20 766f 6c75 7074 6173 2064 6963 6974 m voluptas dicit
000001e0: 7572 2e20 4174 2068 6f63 2069 6e20 656f ur. At hoc in eo
000001f0: 204d 2e20 5265 6665 7274 2074 616d 656e M. Refert tamen
00000200: 2c20 7175 6f20 6d6f 646f 2e20 5175 6964 , quo modo. Quid
00000210: 2073 6571 7561 7475 722c 2071 7569 6420 sequatur, quid
00000220: 7265 7075 676e 6574 2c20 7669 6465 6e74 repugnet, vident
00000230: 2e20 4961 6d20 6964 2069 7073 756d 2061 . Iam id ipsum a
00000240: 6273 7572 6475 6d2c 206d 6178 696d 756d bsurdum, maximum
00000250: 206d 616c 756d 206e 6567 6c65 6769 2e5a malum neglegi.Z
00000260: 0c2c 139e 4bd7 f681 50bc ad7f af40 2ae7 .,..K...P....@*.
00000270: 316b 9b28 bf01 27c8 1b7d 9a49 85a8 596b 1k(..'..}.I..Yk
00000280: 144c e385 361c a064 14f3 583a 5800 0030 .L..6..d..X:X..0

```

Fig 21. xxd result of sample1.txt.conti

xxd를 통해 바이트 수준에서 바이너리의 결과를 관찰하였다. Fig 21를 통해 하이라이트 된 부분에서 Fig 20에서 확인한 값이 정상적으로 덧붙여지는 것을 확인할 수 있다. 또한 0x64 뒤로 특정 바이트 패턴을 가진 값들이 입력되는데 RSA-4096를 사용하는 구조에서 512 바이트를 덧붙이는 로직이 conti 랜섬웨어에 존재한다. 이에 따라 버퍼의 초기화 여부나 처리 여부에 따라 쓰러기 값이 추가되는 것을 확인할 수 있다. 그리고 덧붙여지기 전 값들도 데이터가 손상되지 않고 남아있는 것을 알 수 있다. 이외 파일들도 동일한 흐름으로 출력되기에 본 레포트에서 따로 설명은 생략한다.

5. Evaluation

본 장에서는 3장, 4장에서의 내용을 종합해 분석하며 2장에서 검토한 요구사항들에 대해 평가를 하도록 한다.

Q1) Elfinject를 이용한 바이너리 인젝션을 통해 대칭키 암호 무력화

Q1-1) 매개변수 분석 (평문과 암호문 버퍼)

Q1-2) 암호 연산을 수행하지 않고 input 버퍼에서 output버퍼로 그대로 복사하기

A1-1) 매개변수를 분석해본 결과 평문은 기존 데이터에 32바이트 키와 8바이트 초기화 벡터 값과 1바이트의 구분자로 구성된 페이로드가 RSA 암호를 통해 덧붙여졌다. 이를 통해 ECRYPT_encrypt_bytes에 전달되는 평문의 데이터 버퍼는 원본 데이터 + 512바이트의 암호화된 페이로드로 구성되어 있으며, 암호문 버퍼는 대칭키 암호화가된 암호문 버퍼로 구성되어 있다.

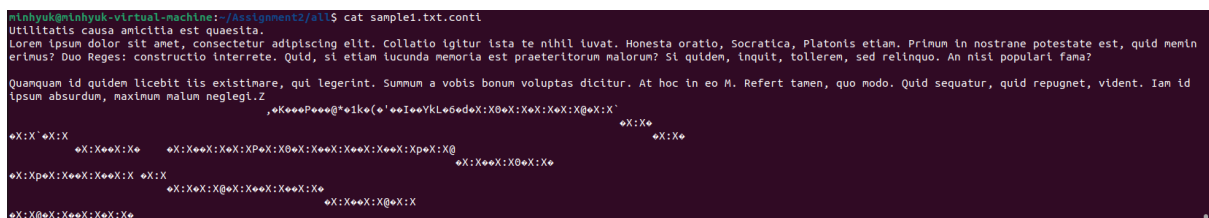


Fig 22. A1-2 Result

A1-2) Fig 13을 통해 ECRYPT_encrypt_bytes 함수가 호출될 때 레지스터 push/pop 연산만 수행하는 섹션을 elfinject로 삽입하여 해당 함수가 호출될 때 암호 함수를 무력화를 하였다. 이에 대한 결과로 Fig 22를 통해 원본 데이터가 손상되지 않음을 알 수 있다.

Q2) LD_PRELOAD를 이용해 동적 라이브러리 overriding을 통해 비대칭키 암호 무력화

Q2-1) 비대칭키 암호 연산 무력화 후 공격된 파일에 삽입된 데이터 무엇인지 확인

A2-1) Fig 12를 통해 원본 데이터 버퍼를 출력 데이터 버퍼로 복사하는 코드로 오버라이딩 하여 비대칭키 암호 연산을 무력화하였다. 이에 대한 결과를 Fig 16에서 확인할 수 있었으며, 정적 분석을 통해 공격된 파일에 삽입된 데이터가 RSA 암호문을 통해 암호화된 데이터라는 것을 알 수 있었다. 구체적으로는 동적 분석을 통해 해당 데이터가 32 바이트 키와 8바이트 초기화 벡터 값을 알 수 있었고, 0x64라는 1바이트의 시그니처 또는 구분자로 삽입된 데이터라는 것을 확인할 수 있었다.

Q3) 랜섬웨어 실행 후 파일이 손상되지 않음을 보일 것

Q3-1)암호화는 되지 않지만 원본 파일의 확장자 변경 및 Ransom note 등이 추가



Fig 23. A3-1 Result - 1

이를 통해 ‘read-append-rename’ 이라는 것을 알 수 있다.

Q5) 랜섬웨어 실행 후 각 공격 대상 파일에 삽입된 페이로드를 분석할 것

Q5-1) 어떻게 획득하였는가?

Q5-2) 암호화가 적용되지 않은 raw data인가?

Q5-3) 크기는 무엇인가?

Q5-4) 무엇인가?

A5-1) 랜섬웨어 실행 후 각 공격 대상 파일에 삽입된 데이터는 RAND_bytes 함수를 통해 획득된 데이터였다.

A5-2) RSA 함수, ECRYPT 함수 무력화 전에는 암호화가 적용된 data 였지만, ECRYPT 함수 무력화 후에도 삽입된 데이터는 RSA 함수에 의해 암호화가 적용되었다. 그러나 RSA 함수도 무력화를 한 후 랜섬웨어를 실행하면 암호화가 적용되지 않은 raw data로 존재한다.

A5-3) 암호화가 적용되기 전에는 512 바이트의 암호 값이 페이로드로 존재하였다. 하지만 암호 함수 무력화 후에는 32바이트+8바이트+1바이트+472바이트로 구분되어 존재한다.

A5-4) 정적 분석과 동적 분석을 통해 해당 페이로드는 구체적으로 32 바이트의 key + 8 바이트의 초기화 벡터 + 1 바이트의 시그니처 + 472 바이트의 쓰레기 값이 라는 것을 알 수 있었다.

6. Conclusion

본 레포트에서는 Conti 랜섬웨어를 분석하기 위해 먼저 2장에서 분석에 필요한 요구사항을 정리하였고, 3장에서 정적 분석과 동적 분석을 통해 랜섬웨어의 내부 구조와 동작 방식을 심층적으로 파악하였다. 또한, 암호화 함수의 무력화를 시도하여 랜섬웨어의 핵심 동작을 제어하는 과정을 수행하였다. 이후 4장에서는 분석 결과를 바탕으로 랜섬웨어의 암호화 동작과 무력화 과정을 확인하고, 5장에서 분석 요구사항에 대한 평가를 통해 분석 작업의 적합성을 검증하였다. 이러한 과정을 통해 Conti 랜섬웨어의 특징과 위협에 대해 구체적으로 이해할 수 있었으며, 랜섬웨어 분석의 기초부터 고급 기술까지 폭넓게 다룰 수 있었다.

Conti 랜섬웨어는 단순히 데이터를 암호화하는 것에 그치지 않고, 네트워크를 통해 확산하며 데이터를 유출하는 이중 갈취 전략을 사용하는 고도화된 악성 소프트웨어이다. 이를 분석하기 위해 ELF 파일 구조, LD_PRELOAD 기법, 어셈블리어의 동작 원리, Stack Convention, C언어 기반의 로직 해석, 역공학 기술 등 다각적인 지식과 기술이 요구되었다. 특히, Conti 랜섬웨어의 암호화

메커니즘을 무력화하고 이를 재현하기 위해 elfinject 기법과 같은 고급 공격 및 분석 기법을 활용하는 과정에서 해당 기술의 중요성과 실무적인 활용성을 체감할 수 있었다.

랜섬웨어 분석 과정은 단순히 코드를 이해하는 것을 넘어선다. 분석 도중 예상과 다른 동작이 발생하거나, 반복적인 디버깅과 테스트를 수행해야 하는 고난도의 작업이 요구되었다. 특히, 분석 중에 얻은 데이터와 결과를 기반으로 랜섬웨어의 복잡한 동작을 재현하는 과정은 고도의 집중력과 문제 해결 능력을 요구하였다. 이번 과제는 악성코드 분석을 처음 시도하는 경험이었지만, 해당 과제를 해결하면서 분석 기법뿐만 아니라 랜섬웨어의 작동 원리에 대한 이해와 함께 악성 프로그램에 대한 접근법을 학습하는 데 성공적인 첫 걸음이 되었다.

결과적으로 본 과제를 통해 악성코드 분석에 대한 자신감을 얻었으며, 추가적인 악성 프로그램 분석에 대한 욕구와 동기를 얻었다. 또한 이번 과제는 난해한 프로그램과 기술적 난관에 직면했을 때에도 해결 가능하다는 자신감을 심어주었고, 분석과 공격 메커니즘에 대한 통찰력을 획득하는 계기가 되었다.