Unpacking the unpleasant FIN7 gift: PackXOR

By Alice Climent-Pommeret

Published: 2024-09-04 · Archived: 2026-04-06 01:10:31 UTC

Published on 4 September, 2024 14min

Summary

In early July 2024, the Sentinel Labs researchers released an extensive article[1] about “FIN7 reboot” tooling, notably introducing “AvNeutralizer”, an anti-EDR tool. This tool has been found in the wild as a packed payload.

In this article, we offer a thorough analysis of the associated private packer that we named “PackXOR”, as well as an unpacking tool. Additionally, while investigating the packer usage, we determined that PackXOR might not be

exclusively leveraged by FIN7.

Background

AvNeutralizer and FIN7

In order to disable EDR (Endpoint Detection and Response) software, AvNeutralizer (also called “AuKill”) relies on vulnerable drivers to terminate EDR related processes from the kernel.

According to Sentinel Labs researchers, AvNeutralizer has been sold since 2022 on “underground” forums such as `xss[.]is`, `exploit[.]in` and “RAMP” by individuals they link with high confidence to the “FIN7” cluster.

Sentinel Labs states that AvNeutralizer can be delivered to targets as a packed or unprotected payload since April 2023, as part of ransomware operations from various threat actors.

Sentinel Labs also notices that “the packer code is identical across various usages, suggesting that FIN7 provides a shared obfuscator to their buyers within the AvNeutralizer bundle”[1].

However, we discovered that PackXOR, the packer for AvNeutralizer, was also used to protect unrelated payloads, such as the “XMRig”[2] cryptominer or XMRig + the “R77 rootkit”[3], which were additionally obfuscated with the open-source “SilentCryptoMiner”[4].

The use of XMRig does not match the known FIN7 TTPs (Tactics, techniques, and procedures). While the packer could still have been used on XMRig payloads to test if it is detected by some security products, we believe such hypothesis is not consistent with the additional use of the SilentCryptoMiner obfuscator.

PackXOR developers might indeed be connected to the FIN7 cluster, but the packer appears to be used for activities that are not related to FIN7.

A catch-up session to packers

In malware analysis, a “packer” is a tool which is used to compress, encrypt, and/or obfuscate a “payload” (which will often be a malicious code).

Packers wrap the original malicious code in “packed data”, and produce a “packed binary” as a result. This packed binary needs to “unpack” packed data before the the payload can be executed:

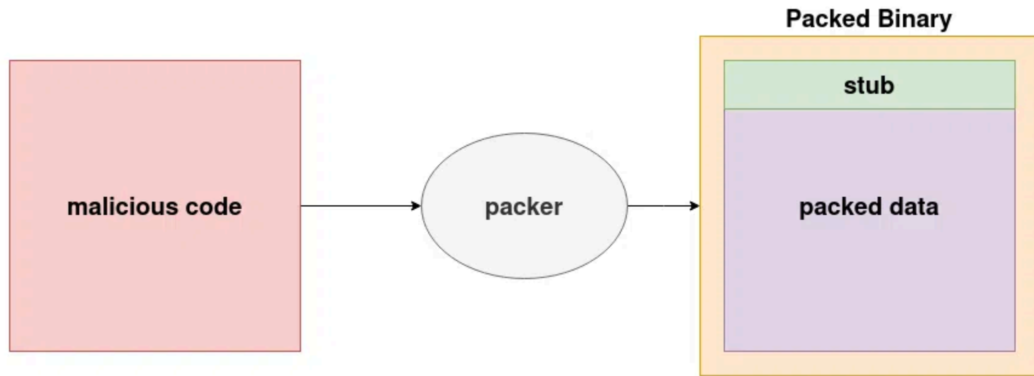


Figure 1 – Packing workflow

Packers' products often contain a decryption stub, which is a small piece of code that is executed first when the packed binary is executed. This stub decrypts and/or decompresses the malicious code (packed data) into its original form, allowing it to execute.

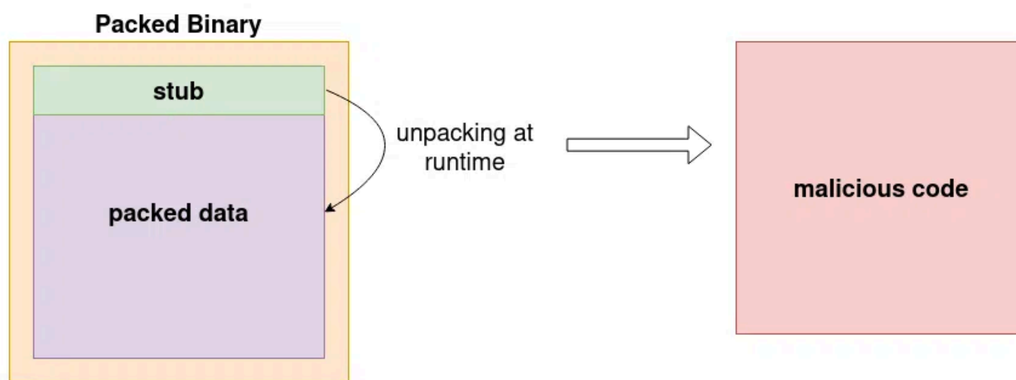


Figure 2 – Unpacking workflow

The aim of packing is to hinder the work of malware analysts and antivirus/EDR software, by concealing payloads and delaying their detection.

PackXOR

Packer logic

Packed data which is produced by the PackXOR packer is structured in 2 sections (see Fig. 3):

- A 40 bytes header that contains:
  - XOR key 1, a XOR key used for a first iteration,
  - the compressed size of the packed payload,
  - the uncompressed size of the packed data,
  - XOR key 2, a XOR key used for a second iteration
- a packed payload.

This packed data is usually found at the begining of the PE `.data` section.

Figure 3 – Data structure of the packed content

In order to conceal the packed payload, and as explained in the Sentinel Labs article[1], the packed binary implements (see Fig. 4):

1. A first XOR iteration (with XOR key 1) on LZNT1 compressed data,
2. A decompression of LZNT1 data,
3. A second XOR iteration (with XOR key 2) on the decompressed data.

```
Dr0 = ExceptionInfo->ContextRecord->Dr0;
uMode = GetLastError();
for ( i = Dr0; i <= (unsigned __int64)(unsigned int)Compressed_Buffer_Size + 0x28; ++i )
{
  Encrypted_Buffer[ExceptionInfo->ContextRecord->Dr0 + i] = (LOBYTE(ExceptionInfo->ContextRecord->Dr6) + first_XOR_key) ^ Encrypted_Buffer[ExceptionInfo->ContextRecord->Dr1 + i];
  one = zero != 3;
  Dr0 += (unsigned int)Factorial_return_last_32bits(2 * Dr0) / one / (1 - ExceptionInfo->ContextRecord->Dr3);
}
LastError = GetLastError();
if ( LastError <= uMode )
  SetLastError(LastError + uMode);
else
  SetLastError(uMode - LastError);
do
{
  if ( FlsAlloc(0i64) != 0xFFFFFFFF )
    Uncompressed_Size = Get_and_Call_RtlDecompressBuffer(
                          (__int64)&Encrypted_Buffer[40],
                          Decompress_buffer,
                          Compressed_Buffer_Size,
                          Uncompress_Buffer_Size);
}
while ( ExceptionInfo->ContextRecord->Dr2 );
if ( !Uncompressed_Size )
  return 0i64;
for ( j = 0; j <= Uncompressed_Size; ++j )
  *(_BYTE *)(Decompress_buffer + (int)j) ^= second_XOR_key;
unpacked_address = AllocateMemory_Copy_Unpacked((char *)Decompress_buffer);
if ( !unpacked_address )
  return 1i64;
```

First XOR iteration / Data decompression / Second XOR iteration

Figure 4 – Unpacking code

In a function of the packer that we called `Get_and_Call_RtlDecompressBuffer` during our analysis, we can see one example of a call to a strings decryption function `decrypt_API_DLL_names` (see Fig. 5). This strings decryption function is described next.

```
__int64 __fastcall Get_and_Call_RtlDecompressBuffer(
        __int64 CompressedBuffer,
        __int64 UncompressedBuffer,
        int CompressedBufferSize,
        unsigned int UncompressedBufferSize)
{
  char *s_RtlDecompressBuffer; // rax
  __int64 v6; // rcx
  unsigned int FinalUncompressedSize; // [rsp+30h] [rbp-18h] BYREF
  int (__fastcall *RtlDecompressBuffer)(__int64, __int64, _QWORD, __int64, int, unsigned int *); // [rsp+38h] [rbp-10h]

  FinalUncompressedSize = 0;
  if ( !hModule_ntdll )
    return 0i64;
  s_RtlDecompressBuffer = decrypt_API_DLL_names(encrypted_RtlDecompressBuffer_string);
  RtlDecompressBuffer = (int (__fastcall *)(__int64, __int64, _QWORD, __int64, int, unsigned int *))EAT_Walk((__int64)hModule_ntdll, (unsigned __int64)s_RtlDecompressBuffer);
  if ( !RtlDecompressBuffer )
    return 0i64;
  LOWORD(v6) = 2;
  if ( RtlDecompressBuffer(
         v6,                               // v6 = COMPRESSION_FORMAT_LZNT1
         UncompressedBuffer,
         UncompressedBufferSize,
         CompressedBuffer,
         CompressedBufferSize,
         &FinalUncompressedSize) >= 0 )
    return FinalUncompressedSize;
  else
    return 0i64;
```

Figure 5 – Call to the “decrypt_api_DLL_names” function

Strings encryption

The packed binary leverages “Run-Time Dynamic Linking” [5] for some specific Windows API functions that it needs to use. The associated required DLLs and Windows API functions names are “encrypted” strings in the packer.

Strings are decrypted just before usage[6] by a dedicated function that we called `decrypt_API_DLL_name`. The “encryption”[7] is implemented using XOR and substraction operations for each byte of a given string (a string being ASCII-encoded):

```
char *__fastcall decrypt_API_DLL_names(char *data)
{
  unsigned __int8 size_string; // [rsp+0h] [rbp-128h]
  char key; // [rsp+2h] [rbp-126h]
  unsigned int i; // [rsp+4h] [rbp-124h]
  char decrypted_string[264]; // [rsp+20h] [rbp-108h] BYREF

  key = *data;
  size_string = data[1];
  for ( i = 0; i < size_string; ++i )
  {
    if ( !data[i + 5] )
    {
      *data = 0;
      break;
    }
    decrypted_string[i] = (data[i + 5] ^ key) - i - 1;
  }
  decrypted_string[size_string] = 0;
  strcpy(data, decrypted_string);
  return data;
}
```

Figure 6 – Strings encryption function

Encrypted strings are stored in “data blobs” which match a specific layout:

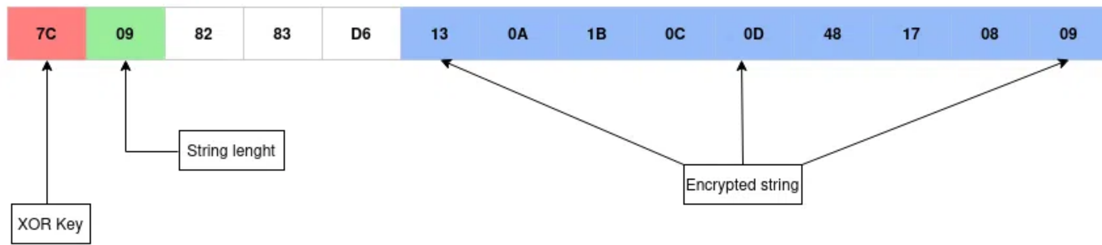


Figure 7 – Structure of a blob for an encrypted string

The first byte of the blob (in red) is the `XOR` key which is used for the byte by byte “encryption”.

The second byte (in green) contains the string length in bytes. Between the string length and the first byte of the encrypted string, 3 bytes are unused.

In the packed binary, encrypted data blobs are stored one after the others. The color code used is the same that the one in the illustration below. The non-colored bytes are unused:

+Po…on-	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000000h	00	00	00	00	00	00	00	00	D9	17	27	26	73	96	AF	9D
000010h	A9	A8	AC	B3	B0	A4	B6	B8	AC	A6	5B	5D	A8	A4	86	A1
000020h	58	5D	51	49	00	07	76	66	6D	71	69	79	75	6F	00	00
000030h	0B	13	F5	F4	A1	58	7D	64	4C	7C	60	63	77	65	54	75
000040h	7A	74	69	7C	89	7D	78	7C	00	07	72	66	79	63	6C	6F
000050h	63	68	00	00	00	00	00	00	AB	13	55	54	01	F8	DD	C4
000060h	E3	C1	C2	DD	DE	D2	D7	DB	D4	2B	FB	2F	DD	DC	DC	2E
000070h	00	07	65	77	76	71	69	64	6C	6F	71	6B	70	78	66	00
000080h	AB	0C	55	54	01	C7	CC	DE	D9	C1	D9	91	91	9C	C5	DC
000090h	D3	00	07	65	63	76	78	77	72	6A	00	00	00	00	00	00
0000A0h	38	1B	C6	C7	92	6C	5F	4F	61	4B	56	50	4E	55	4E	48
0000B0h	48	6A	BE	4A	4D	B9	BE	44	BB	BB	64	B8	BC	B5	47	B5
0000C0h	00	07	73	6C	65	6B	75	64	66	7A	71	6E	76	67	67	00
0000D0h	6D	16	93	92	C7	22	1B	3E	1B	19	17	01	06	10	0D	19
0000E0h	13	EC	EE	1D	11	33	1A	ED	EE	EA	E2	00	07	64	74	78
0000F0h	68	6E	69	6E	00	00	00	00	7C	09	82	83	D6	13	0A	1B
000100h	0C	0D	48	17	08	09	00	07	6C	6C	7A	73	6E	6D	6C	62
000110h	78	72	69	7A	69	00	00	00	00	00	00	00	00	00	00	00

Figure 8 – Encrypted data blobs in the packer

As an example, let’s decrypt the last encrypted data blob that is shown in the screenshot above (see Fig. 8). Here, the XOR key is `0x7C`. If we follow the “decryption” routine and for each byte of the data blob, we need to: XOR the byte with the key, then substract the current byte position index (in data blob) minus 1 to the result.

```
 ((13 xor 7C) - 0) - 1 = 6F - 1 = 6E = n
((0A xor 7C) - 1) - 1 = 75 - 1 = 74 = t
((1B xor 7C) - 2) - 1 = 65 - 1 = 64 = d
((0C xor 7C) - 3) - 1 = 6D - 1 = 6C = l
((0D xor 7C) - 4) - 1 = 6D - 1 = 6C = l
((48 xor 7C) - 5) - 1 = 2F - 1 = 2E = .
```

```
((17 xor 7C) - 6) - 1 = 65 - 1 = 64 = d
((08 xor 7C) - 7) - 1 = 6D - 1 = 6C = l
((09 xor 7C) - 8) - 1 = 6D - 1 = 6C = l
```

The `XOR` key is different for almost every string in a given binary sample, and changes with every sample. Changing the `XOR` keys between strings and samples increase the odds of bypassing a static analysis.

PackXOR usage

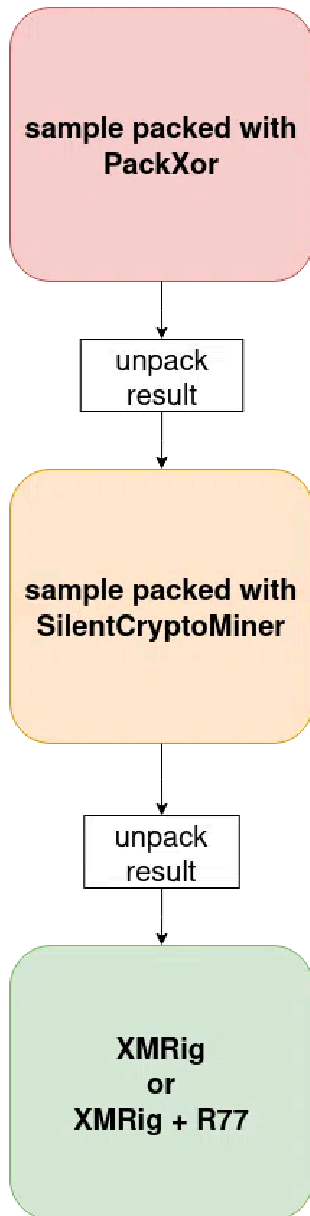
During our research, we could identify 4 different additional payloads (other than AvNeutralizer) that we believe with medium to high confidence were packed with PackXOR, because the unpacking code is identical in all samples.

Three of the identified samples drop the XMRig[2] cryptominer or XMRig + the R77 rootkit[3]. Between those final payloads and PackXOR-produced code, we discovered a second and sometimes third layer of obfuscation (see Fig. 9):

- some payloads (SHA-256 `e3505901fd44c8f6597ca9c512375b6ecbf3dc21dbae3d373318c99929d62091` and `b86612a6d62a1789031248bdb732b8bff51acaeaa687c3559f0980560a8abf2f`) were packed with the open-source SilentCryptoMiner[4] obfuscator ,
- a payload (SHA-256 `cf1d985a33b39d332d4bac33d971a004dcd18cea82ff1b291c6a5046e073414d`) which was obfuscated with SilentCryptoMiner was additionnally obfuscated with a “commercial” packing tool (Hidden Malware Builder[8]).

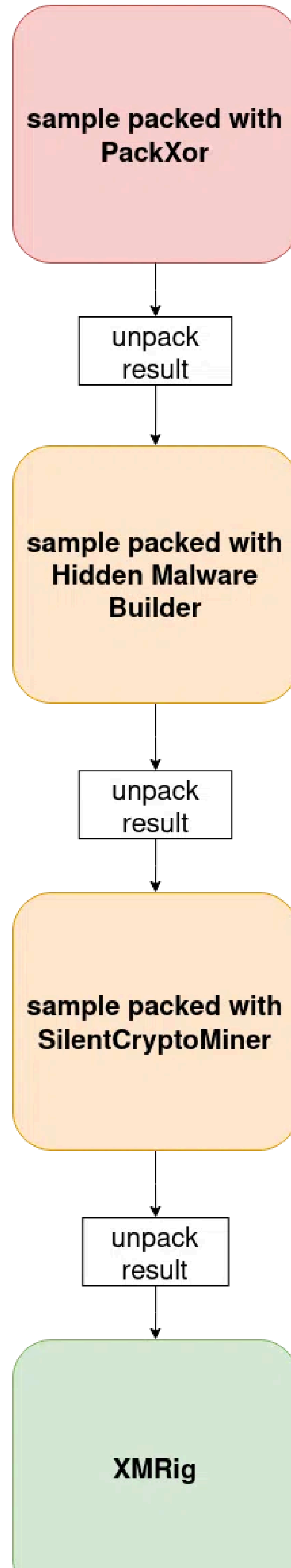
Samples:

- e3505*
- b8661*



Sample:

- cf1d9*



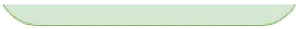


Figure 9 – Layers of obfuscation

One of the packed binary samples we identified (SHA-256 `632b068e1b8fbc54eb0b30f01455c73396deb5f8e3bbd3b171fb69b6936a6019`) dropped another type of payload, which is very similar to a data exfiltration tool that was documented in an article from ReversingLab in 2021[9].

Unpacker

According to Sentinel Labs and following our own research, it appears PackXOR is used by different ransomware operators, and to pack different tools. As a result we thought that providing an unpacker could be of use to the cybersecurity community.

We developed one which can be downloaded from our Github repository.

```
usage: packxor_unpacker.py [-h] [--file FILE] [--offset OFFSET]

Unpacker for PackXOR

options:
  -h, --help       show this help message and exit
  --file FILE      Packed PackXOR Malware
  --offset OFFSET  Optional. Offset of the packed header (in hexadecimal). No prefix (0x, x, etc)
```

If you already know the offset of the packed data structure header in the binary you want to unpack, you can pass it directly with the `--offset` argument.

```
$ python packxor_unpacker.py --file 050637.exe --offset 1a00
XOR key for first iteration : 0x1f
XOR key for second iteration : 0x4f
Size of compressed data (in bytes): 62958
Size of uncompressed data (in bytes): 80896
Unpacking SUCCESS
Unpacked file available in 050637_unpacked.exe
```

However, if you don't have time or don't want to reverse the binary to find such offset, no worries! Without `--offset`, the script will try to automatically the header offset and unpack the data.

```
$ python packxor_unpacker.py --file 050637.exe
Offset header not provided as an argument. Trying to find it anyway.
Packer header found
XOR key for first iteration : 0x1f
XOR key for second iteration : 0x4f
```

```
Size of compressed data (in bytes): 62958
Size of uncompressed data (in bytes): 80896
Unpacking SUCCESS
Unpacked file available in 050637_unpacked.exe
```

Appendix

Indicators of compromise (IOCs)

Associated IOCs are also available on our GitHub repository.

Hashes (SHA-256)

Packed

```
 0506372e2c2b6646c539ac5a08265dd66d0da58a25545e444c25b9a02f8d9a44|AvNeutralizer
146c68ca89b8b0378c2c6fb978892aace0235c7038879e85b3764556b0dbf2a5|AvNeutralizer
cf1d985a33b39d332d4bac33d971a004dcd18cea82ff1b291c6a5046e073414d|XMRig (packed with: PackXOR+Hidden Malware Bui
e3505901fd44c8f6597ca9c512375b6ecbf3dc21dbae3d373318c99929d62091|XMRig (packed with: PackXOR+SilentCryptoMiner)
b86612a6d62a1789031248bdb732b8bff51acaeaa687c3559f0980560a8abf2f|XMRig+R77 (packed with: PackXOR+SilentCryptoMi
dcc7fd38fced82cc04cb6fa0d189d2924163494e542f6c516e6588c110ab7554|Data exfiltrator/bot (packed with: PackXOR)
```

Unpacked

```
 f15e6ff7f1ba8f7aad1adb88300a5ea367d6b5388f41d602f978d2885aa2ed38|AvNeutralizer
56af567979acaec20bab9a36064ee5f31b96fceaa5487f6ba2db9ff6360d9a51|AvNeutralizer
40a8ffc5bbcb3befc90f269e32ab96b3ff32768f1fc0317a00f86f9b1161cdeb|XMRig+R77 (packed with: SilentCryptoMiner)
42ca0d62a9516cbf4a1ffcd9097d2f2c3b135f82b1c07adf586ef5b23ce96197|XMRig (packed with: Hidden Malware Builder+Sile
1428e14c9c86e8f068e37efc11190ee16f2cdb9bc808308c5450389ee2893c10|XMRig (packed with: SilentCryptoMiner)
632b068e1b8fbc54eb0b30f01455c73396deb5f8e3bbd3b171fb69b6936a6019|Data exfiltrator/bot
```

Yara rule

```
 rule PackXOR
{
    meta:
        description = "Detection rule for PackXOR"
        references = "https://harfanglab.io/insidethelab/unpacking-packxor/"
        hash = "0506372e2c2b6646c539ac5a08265dd66d0da58a25545e444c25b9a02f8d9a44"
        date = "2024-08-05"
        author = "Harfanglab"
        context = "file"
    strings:
        $s_packer_xor = {
```

```
        4? 63 [3]                    // movsxd  rax, dword [rsp+0x50 {var_78}]
        4? 8b [2-6]                  // mov     rcx, qword [rsp+0xd0 {arg_8}]
        4? 8b [2-6]                  // mov     rcx, qword [rcx+0x8]
        4? 0? [2]                    // add     rax, qword [rcx+0x50]
        4? 8d [5]                    // lea     rcx, [rel data_140003020]
        0f (b6|b7) [1-5]             // movzx   eax, byte [rcx+rax]
        0f (b6|b7) [1-5]             // movzx   ecx, byte [rel data_14002399c]
        4? 8b [2-6]                  // mov     rdx, qword [rsp+0xd0 {arg_8}]
        4? 8b [2-6]                  // mov     rdx, qword [rdx+0x8]
        4? 0? [2]                    // add     rcx, qword [rdx+0x68]
        0f (b6|b7) [1-5]             // movzx   ecx, cl
        33 ??                        // xor     eax, ecx
        4? 63 [3]                    // movsxd  rcx, dword [rsp+0x50 {var_78}]
        4? 8b [2-6]                  // mov     rdx, qword [rsp+0xd0 {arg_8}]
        4? 8b [2-6]                  // mov     rdx, qword [rdx+0x8]
        4? 0? [2]                    // add     rcx, qword [rdx+0x48]
        4? 8d [5]                    // lea     rdx, [rel data_140003020]
        88 04 0a                     // mov     byte [rdx+rcx], al
        0f (b6|b7)                   // movzx   eax, byte [rel data_14000301e]
    }
    $s_packer_decrypt_conf = {
        8b [1-3]            // mov     eax, dword [rsp+0x4 {i}]
        ff ??               // inc     eax
        89 [1-3]            // mov     dword [rsp+0x4 {i}], eax
        0f b6 [1-3]         // movzx   eax, byte [rsp {var_128}]
        39 [1-3]            // cmp     dword [rsp+0x4 {i}], eax
        73 ??               // jae     0x140001d59
        8b [1-3]            // mov     eax, dword [rsp+0x4 {i}]
        83 ?? 05            // add     eax, 0x5
        8b ??               // mov     eax, eax
        4? 8b [2-6]         // mov     rcx, qword [rsp+0x130 {arg_8}]
        0f be [1-3]         // movsx   eax, byte [rcx+rax]
        85 ??               // test    eax, eax
        74 ??               // je      0x140001d40
        0f b6 [1-3]         // movzx   eax, byte [rsp+0x2 {var_126}]
        8b [3]              // mov     ecx, dword [rsp+0x4 {i}]
        83 ?? 05            // add     ecx, 0x5
        8b ??               // mov     ecx, ecx
        4? 8b [4-6]         // mov     rdx, qword [rsp+0x130 {arg_8}]
        0f (be|bf) [1-3]    // movsx   ecx, byte [rdx+rcx]
        33 ??               // xor     eax, ecx
        2b [1-3]            // sub     eax, dword [rsp+0x4 {i}]
        ff ??               // dec     eax
        8b [1-3]            // mov     ecx, dword [rsp+0x4 {i}]
        88 [1-3]            // mov     byte [rsp+rcx+0x20 {var_108}], al
        eb ??               // jmp     0x140001d57
        b8 01 00 00 00      // mov     eax, 0x1
```

```
        4? 6b ?? 00         // imul    rax, rax, 0x0
        4? 8b [4-6]         // mov     rcx, qword [rsp+0x130 {arg_8}]
        c6 [1-3] 00         // mov     byte [rcx+rax], 0x0
        eb ??               // jmp     0x140001d59
        eb                  // jmp     0x140001ce7
    }
    $s_packer_find_entry_point = {
        4? 63 [1-4]             // movsxd  rax, dword [rsp {var_38_1}]
        4? 3b [1-4]             // cmp     rax, qword [rsp+0x20 {var_18_1}]
        73 ??                   // jae     0x140001c7f
        48 8b [1-4]             // mov     rax, qword [rsp+0x10 {var_28_1}]
        0f b7 [1-4]             // movzx   eax, word [rax]
        c1 ?? 0c                // sar     eax, 0xc
        83 ?? 0a                // cmp     eax, 0xa
        75 ??                   // jne     0x140001c7d
        4? 8b [1-4]             // mov     rax, qword [rsp+0x8 {var_30}]
        8b [1-4]                // mov     eax, dword [rax]
        4? 03 [1-4]             // add     rax, qword [rsp+0x40 {arg_8}]
        4? 8b [1-4]             // mov     rcx, qword [rsp+0x10 {var_28_1}]
        0f b7 [1-4]             // movzx   ecx, word [rcx]
        81 ?? ff 0f 00 00       // and     ecx, 0xfff
        4? 63 [1-4]             // movsxd  rcx, ecx
        4? 03 [1-4]             // add     rax, rcx
        4? 89 [1-4]             // mov     qword [rsp+0x18 {var_20_1}], rax
        4? 8b [1-4]             // mov     rax, qword [rsp+0x18 {var_20_1}]
        4? 8b [1-4]             // mov     rax, qword [rax]
        4? 03 [1-4]             // add     rax, qword [rsp+0x50 {arg_18}]
        4? 8b [1-4]             // mov     rcx, qword [rsp+0x18 {var_20_1}]
        4? 89 [1-4]             // mov     qword [rcx], rax
        eb 93                   // jmp     0x140001c12
    }
    $s_packer_find_entry_point_rtlcreateuserthtread = {
        4? 8b [1-4]                // mov     rax, qword [rsp+0x70 {var_58_1}]
        8b [1-4]                   // mov     eax, dword [rax+0x28]
        4? 03 [1-4]                // add     rax, qword [rsp+0x68 {var_60_1}]
        4? 89 [2-6]                // mov     qword [rsp+0x88 {var_40_1}], rax
        ff [2-6]                   // call    qword [rsp+0x88 {var_40_1}]
        4? 8d [2-6]                // lea     rax, [rsp+0x9c {var_2c}]
        4? 89 [1-4]                // mov     qword [rsp+0x48 {var_80_1}], rax {var_2c}
        4? 8d [2-6]                // lea     rax, [rsp+0xb8 {var_10}]
        4? 89 [1-4]                // mov     qword [rsp+0x40 {var_88_1}], rax {var_10}
        4? c7 [3-7]                // mov     qword [rsp+0x38 {var_90}], 0x0
        4? 8b [2-6]                // mov     rax, qword [rsp+0x88 {var_40_1}]
        4? 89 [1-4]                // mov     qword [rsp+0x30 {var_98_1}], rax
        4? c7 [3-7]                // mov     qword [rsp+0x28 {var_a0}], 0x0
        4? c7 [3-7 ]               // mov     qword [rsp+0x20 {var_a8}], 0x0
        4? 33 ??                   // xor     r9d, r9d  {0x0}
```

```
            4? ?? 01                    // mov     r8b, 0x1
            33 ??                       // xor     edx, edx  {0x0}
            4? c? ?? ff ff ff ff        // mov     rcx, 0xffffffffffffffff
            ff                          // call    qword [rsp+0xa0 {var_28_1}]
        }
        $s_packer_string_encryption = {
            0f B? [1-2]      // movzx   eax, [rsp+128h+size_string]
            39 [1-3]         // cmp     [rsp+128h+var_124], eax
            73 ??            // jnb     short loc_140001CC9
            8B [1-3]         // mov     eax, [rsp+128h+var_124]
            83 ?? 05         // add     eax, 5
            8B ??            // mov     eax, eax
            4? 8B [1-6]      // mov     rcx, [rsp+128h+arg_0]
            0F B? [1-2]      // movsx   eax, byte ptr [rcx+rax]
            85 ??            // test    eax, eax
            74 ??            // jz      short loc_140001CB0
            0f B? [1-3]      // movzx   eax, [rsp+128h+key]
            8B [1-3]         // mov     ecx, [rsp+128h+var_124]
            83 ?? 05         // add     ecx, 5
            8B ??            // mov     ecx, ecx
            4? 8B [1-6]      // mov     rdx, [rsp+128h+arg_0]
            0F B? [1-2]      // movsx   ecx, byte ptr [rdx+rcx]
            33 ??            // xor     eax, ecx
            2B [1-3]         // sub     eax, [rsp+128h+var_124]
            FF ??            // dec     eax
            8B [1-3]         // mov     ecx, [rsp+128h+var_124]
            88 [1-3]         // mov     [rsp+rcx+128h+decrypted_string], al
            EB               // jmp     short loc_140001CC7
        }
    condition:
        uint16(0) == 0x5A4D
        and uint32(uint32(0x3C)) == 0x00004550
        and filesize < 20MB
        2 of ($s_packer*)
}
```

---

1. https://www.sentinelone.com/labs/fin7-reboot-cybercrime-gang-enhances-ops-with-new-edr-bypasses-and-automated-attacks/ ↩ ↩ ↩

2. https://github.com/xmrig/xmrig ↩ ↩

3. https://bytecode77.com/r77-rootkit ↩ ↩

4. https://github.com/SilentCryptoMiner/SilentCryptoMiner ↩ ↩

5. `Run-time dynamic linking` is a way to load DLLs (Dynamic Link Library) and import functions from them only when needed, rather than at the executable startup. This process involves the Windows API functions `GetModuleHandle`, `LoadLibrary`, and `GetProcAddress`. Malware often uses `Run-time dynamic linking` in order to evade detection from static analysis tools. ↩
6. Strings are “decrypted” just before usage in `LoadLibrary` and `GetProcAddress` functions. ↩
7. https://archive.org/details/flooved1478/page/n1/mode/2up ↩
8. https://poison.tools/product/poison-fud-crypter/ ↩
9. https://www.reversinglabs.com/blog/data-exfiltrator ↩

Source: https://harfanglab.io/insidethelab/unpacking-packxor/