BendyBear: Novel Chinese Shellcode Linked With Cyber Espionage Group BlackTech

unit42.paloaltonetworks.com/bendybear-shellcode-blacktech

February 9, 2021

By Mike Harbison

February 9, 2021 at 3:00 AM

Category: Unit 42

Tags: BendyBear, BlackTech, DbgPrint, malware, shellcode, TaiDoor, WaterBear

This post is also available in: 日本語 (Japanese)

Executive Summary

Highly malleable, highly sophisticated and over 10,000 bytes of machine code. This is what Unit 42 researchers were met with during code analysis of this “bear” of a file. The code behavior and features strongly correlate with that of the WaterBear malware family, which has been active since as early as 2009. Analysis by Trend Micro and TeamT5 unveiled WaterBear as a multifaceted, stage-two implant, capable of file transfer, shell access, screen capture and much more. The malware is associated with the cyber espionage group BlackTech, which many in the broader threat research community have assessed to have ties to the Chinese government, and is believed to be responsible for recent attacks against several East Asian government organizations. Due to the similarities with WaterBear, and the polymorphic nature of the code, Unit 42 named this

novel Chinese shellcode “BendyBear.” It stands in a class of its own in terms of being one of the most sophisticated, well-engineered and difficult-to-detect samples of shellcode employed by an Advanced Persistent Threat (APT).

The BendyBear sample was determined to be x64 shellcode for a stage-zero implant whose sole function is to download a more robust implant from a command and control (C2) server. Shellcode, despite its name, is used to describe the small piece of code loaded onto the target immediately following exploitation, regardless of whether or not it actually spawns a command shell. At 10,000+ bytes, BendyBear is noticeably larger than most, and uses its size to implement advanced features and anti-analysis techniques, such as modified RC4 encryption, signature block verification, and polymorphic code.

The sample analyzed in this blog was identified by its connections to a malicious C2 domain published by Taiwan’s Ministry of Justice Investigation Bureau in August 2020. It was discovered absent additional information regarding the exploit vector, potential victims or intended use.

Palo Alto Networks customers can be protected from the attacks outlined in this blog with the Next-Generation Firewall alongside DNS Security, URL Filtering and WildFire security subscriptions, and Cortex XDR.

A New Class of Shellcode

At a macro level, BendyBear is unique in that it:

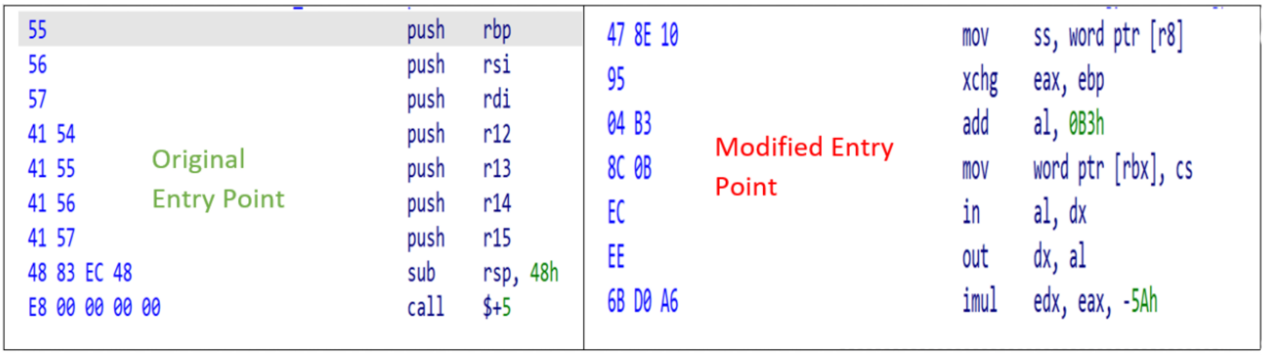
- Transmits payloads in modified RC4-encrypted chunks. This hardens the encryption of the network communication, as a single RC4 key will not decrypt the entire payload.
- Attempts to remain hidden from cybersecurity analysis by explicitly checking its environment for signs of debugging.
- Leverages existing Windows registry key that is enabled by default in Windows 10 to store configuration data.
- Clears the host’s DNS cache every time it attempts to connect to its C2 server, thereby requiring that the host resolve the current IP address for the malicious C2 domain each time.
- Generates unique session keys for each connection to the C2 server.
- Obscures its connection protocol by connecting to the C2 server over a common port (443), thereby blending in with normal SSL network traffic.
- Employs polymorphic code, changing its runtime footprint during code execution to thwart memory analysis and evade signaturing.
- Encrypts or decrypts function blocks (code blocks) during runtime, as needed, to evade detection.
- Uses position independent code (PIC) to throw off static analysis tools.

In the following sections, we provide an in-depth technical breakdown of each of these capabilities.

Technical Details

Shellcode Execution

The shellcode (SHA256: 64CC899EC85F612270FCFB120A4C80D52D78E68B05CAF1014D2FE06522F1E2D0) is considered to be a stager, or downloader, whose function is to download an implant from a C2 server. During execution, the code employs byte randomization to obscure its behavior. This is achieved by using the host's current time as a seed for a pseudorandom number generator, and then performing additional operations against that output. The resulting values are used to overwrite blocks of previously executed code. This byte manipulation is the first anti-analysis technique observed in the code, as any attempt to dump the memory segment would result in illegitimate or incorrect operations. Figure 1 shows an example of the shellcode main entry point before and during runtime execution.



```
55                 push    rbp
56                 push    rsi
57                 push    rdi
41 54              push    r12
41 55              push    r13
41 56              push    r14
41 57              push    r15
48 83 EC 48        sub     rsp, 48h
E8 00 00 00 00     call    $+5
```

```
47 8E 10           mov     ss, word ptr [r8]
95                 xchg    eax, ebp
04 B3              add     al, 0B3h
8C 0B              mov     word ptr [rbx], cs
EC                 in      al, dx
EE                 out     dx, al
6B D0 A6           imul    edx, eax, -5Ah
```

Figure 1. Modified shellcode runtime example.

Because shellcode lacks the ability to run on its own, a Windows loader is required to allocate an environment in memory for it to execute. At the time of analysis, no loader had been identified for this shellcode; Therefore, Unit 42 created a custom loader to study the code during its runtime execution. Since then, however, several older installers were discovered with embedded WaterBear shellcode based on attributes identified from this sample. More information on these loaders can be found in the Appendix section "x86 WaterBear Loaders".

The shellcode begins by locating the target's Process Environment Block (PEB) to check if it's currently being debugged. However, the code is written such that it pulls both the "BeingDebugged" and "BitField" values from the PEB, resulting in code logic that invalidates the debugger check. Because of this, the shellcode will always fail to recognize when a debugger is attached. This routine is performed 52 times in a while loop.

Next, the shellcode iterates through the PEB's loader module list looking for the base address of Kernel32.dll. This is typical of shellcode, as the Kernel32.dll base address is necessary to resolve any dependency files required by the shellcode to run. With this address, the shellcode loads its dependency modules and resolves any necessary Windows Application Programming Interface (API) calls using standard shellcode API hashing. The following modules are loaded:

- Advapi32.dll

- Kernel32.dll
- Msvcrt.dll
- User32.dll
- Ws2_32.dll

With the shellcode initialization complete, it moves onto its main function. It begins by querying the target's registry, at the following key:

> HKEY_CURRENT_USER\Console\QuickEdit

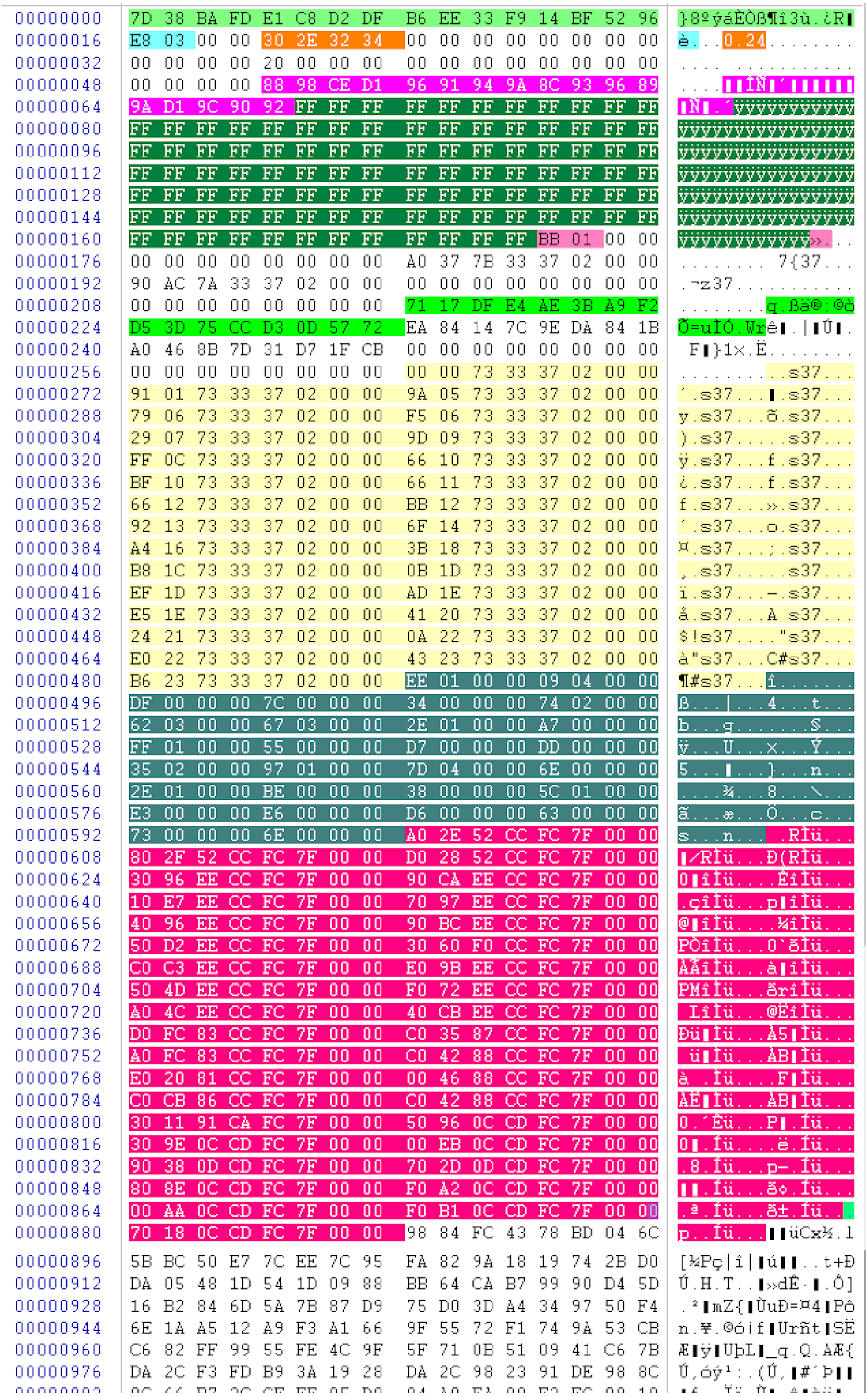
This registry key is used by the Windows command prompt to enable Quick Edit mode. Quick Edit mode allows copy and paste from the command prompt to the clipboard. By default, this key contains a REG_DWORD, a 32-bit number of either 1 for enabled or 0 for disabled. BendyBear reads this value, multiplies it by 1000 and performs the following calculation on the result:

If the result is less than 1,000 or greater than 3,300,000 the shellcode configuration (QuickEdit) is 4,000 (0xFA0) otherwise it is the result of the computed value.

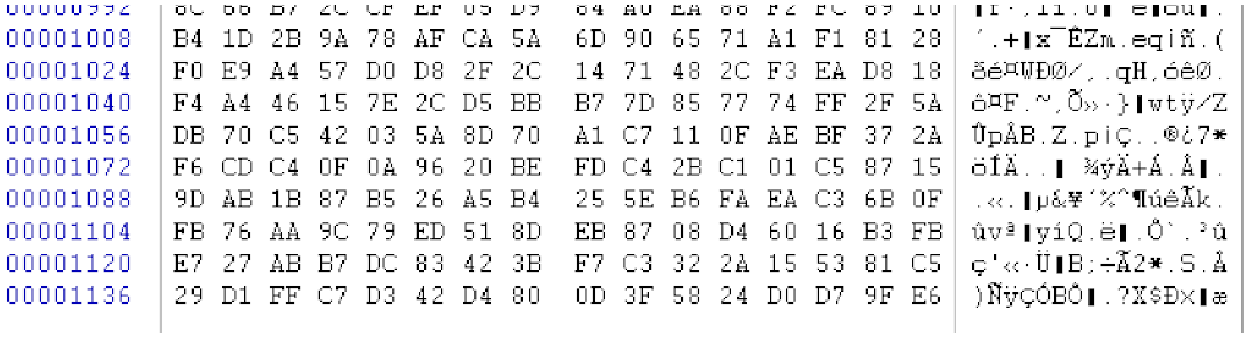
Refer to the highlighted light blue value in Figure 2 Shellcode configuration.

This check is performed each and every time the shellcode is executed. One explanation for the use of this key is that the value is written to by the shellcode loader (to a value other than 0 or 1) and it's used by the shellcode to obtain configuration settings.

It then decrypts its internal configuration structure, which is 1,152 bytes. An example is shown in Figure 2.



```
00000000  7D 38 BA FD E1 C8 D2 DF  B6 EE 33 F9 14 BF 52 96  }8ºýáÈÒß¶î3ù.¿R▌
00000016  E8 03 00 00 30 2E 32 34  00 00 00 00 00 00 00 00  è...0.24........
00000032  00 00 00 00 20 00 00 00  00 00 00 00 00 00 00 00  .... ...........
00000048  00 00 00 00 88 98 CE D1  96 91 94 9A 8C 93 96 89  ....▌▌ÎÑ▌´▌▌▌▌▌▌
00000064  9A D1 9C 90 92 FF FF FF  FF FF FF FF FF FF FF FF  ▌Ñ▌.´ÿÿÿÿÿÿÿÿÿÿÿ
00000080  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000096  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000112  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000128  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000144  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000160  FF FF FF FF FF FF FF FF  FF FF FF FF BB 01 00 00  ÿÿÿÿÿÿÿÿÿÿÿÿ»...
00000176  00 00 00 00 00 00 00 00  A0 37 7B 33 37 02 00 00  ......... 7{37...
00000192  90 AC 7A 33 37 02 00 00  00 00 00 00 00 00 00 00  .¬z37...........
00000208  00 00 00 00 00 00 00 00  71 17 DF E4 AE 3B A9 F2  ........q.ßä®;©ò
00000224  D5 3D 75 CC D3 0D 57 72  EA 84 14 7C 9E DA 84 1B  Õ=uÌÓ.Wrê▌.|▌Ú▌.
00000240  A0 46 8B 7D 31 D7 1F CB  00 00 00 00 00 00 00 00   F▌}1×.Ë........
00000256  00 00 00 00 00 00 00 00  00 00 73 33 37 02 00 00  ..........s37...
00000272  91 01 73 33 37 02 00 00  9A 05 73 33 37 02 00 00  ´.s37...▌.s37...
00000288  79 06 73 33 37 02 00 00  F5 06 73 33 37 02 00 00  y.s37...õ.s37...
00000304  29 07 73 33 37 02 00 00  9D 09 73 33 37 02 00 00  ).s37.....s37...
00000320  FF 0C 73 33 37 02 00 00  66 10 73 33 37 02 00 00  ÿ.s37...f.s37...
00000336  BF 10 73 33 37 02 00 00  66 11 73 33 37 02 00 00  ¿.s37...f.s37...
00000352  66 12 73 33 37 02 00 00  BB 12 73 33 37 02 00 00  f.s37...».s37...
00000368  92 13 73 33 37 02 00 00  6F 14 73 33 37 02 00 00  ´.s37...o.s37...
00000384  A4 16 73 33 37 02 00 00  3B 18 73 33 37 02 00 00  ¤.s37...;.s37...
00000400  B8 1C 73 33 37 02 00 00  0B 1D 73 33 37 02 00 00  ¸.s37.....s37...
00000416  EF 1D 73 33 37 02 00 00  AD 1E 73 33 37 02 00 00  ï.s37...-.s37...
00000432  E5 1E 73 33 37 02 00 00  41 20 73 33 37 02 00 00  å.s37...A s37...
00000448  24 21 73 33 37 02 00 00  0A 22 73 33 37 02 00 00  $!s37...."s37...
00000464  E0 22 73 33 37 02 00 00  43 23 73 33 37 02 00 00  à"s37...C#s37...
00000480  B6 23 73 33 37 02 00 00  EE 01 00 00 09 04 00 00  ¶#s37...î.......
00000496  DF 00 00 00 7C 00 00 00  34 00 00 00 74 02 00 00  ß...|...4...t...
00000512  62 03 00 00 67 03 00 00  2E 01 00 00 A7 00 00 00  b...g.......§...
00000528  FF 01 00 00 55 00 00 00  D7 00 00 00 DD 00 00 00  ÿ...U...×...Ý...
00000544  35 02 00 00 97 01 00 00  7D 04 00 00 6E 00 00 00  5...▌...}...n...
00000560  2E 01 00 00 BE 00 00 00  38 00 00 00 5C 01 00 00  ....¾...8...\...
00000576  E3 00 00 00 E6 00 00 00  D6 00 00 00 63 00 00 00  ã...æ...Ö...c...
00000592  73 00 00 00 6E 00 00 00  A0 2E 52 CC FC 7F 00 00  s...n... .RÌü...
00000608  80 2F 52 CC FC 7F 00 00  D0 28 52 CC FC 7F 00 00  ▌/RÌü...Ð(RÌü...
00000624  30 96 EE CC FC 7F 00 00  90 CA EE CC FC 7F 00 00  0▌îÌü....ÊîÌü...
00000640  10 E7 EE CC FC 7F 00 00  70 97 EE CC FC 7F 00 00  .çîÌü...p▌îÌü...
00000656  40 96 EE CC FC 7F 00 00  90 BC EE CC FC 7F 00 00  @▌îÌü....¼îÌü...
00000672  50 D2 EE CC FC 7F 00 00  30 60 F0 CC FC 7F 00 00  PÒîÌü...0`ðÌü...
00000688  C0 C3 EE CC FC 7F 00 00  E0 9B EE CC FC 7F 00 00  ÀÃîÌü...à▌îÌü...
00000704  50 4D EE CC FC 7F 00 00  F0 72 EE CC FC 7F 00 00  PMîÌü...ðrîÌü...
00000720  A0 4C EE CC FC 7F 00 00  40 CB EE CC FC 7F 00 00   LîÌü...@ËîÌü...
00000736  D0 FC 83 CC FC 7F 00 00  C0 35 87 CC FC 7F 00 00  Ðü▌Ìü...À5▌Ìü...
00000752  A0 FC 83 CC FC 7F 00 00  C0 42 88 CC FC 7F 00 00   ü▌Ìü...ÀB▌Ìü...
00000768  E0 20 81 CC FC 7F 00 00  00 46 88 CC FC 7F 00 00  à .Ìü....F▌Ìü...
00000784  C0 CB 86 CC FC 7F 00 00  C0 42 88 CC FC 7F 00 00  ÀË▌Ìü...ÀB▌Ìü...
00000800  30 11 91 CA FC 7F 00 00  50 96 0C CD FC 7F 00 00  0.´Êü...P▌.Íü...
00000816  30 9E 0C CD FC 7F 00 00  00 EB 0C CD FC 7F 00 00  0▌.Íü....ë.Íü...
00000832  90 38 0D CD FC 7F 00 00  70 2D 0D CD FC 7F 00 00  .8.Íü...p-.Íü...
00000848  80 8E 0C CD FC 7F 00 00  F0 A2 0C CD FC 7F 00 00  ▌▌.Íü...ð¢.Íü...
00000864  00 AA 0C CD FC 7F 00 00  F0 B1 0C CD FC 7F 00 00  .ª.Íü...ð±.Íü...
00000880  70 18 0C CD FC 7F 00 00  98 84 FC 43 78 BD 04 6C  p..Íü...▌▌üCx½.l
00000896  5B BC 50 E7 7C EE 7C 95  FA 82 9A 18 19 74 2B D0  [¼Pç|î|▌ú▌▌..t+Ð
00000912  DA 05 48 1D 54 1D 09 88  BB 64 CA B7 99 90 D4 5D  Ú.H.T..▌»dÊ·▌.Ô]
00000928  16 B2 84 6D 5A 7B 87 D9  75 D0 3D A4 34 97 50 F4  .²▌mZ{▌ÙuÐ=¤4▌Pô
00000944  6E 1A A5 12 A9 F3 A1 66  9F 55 72 F1 74 9A 53 CB  n.¥.©ó¡f▌Urñt▌SË
00000960  C6 82 FF 99 55 FE 4C 9F  5F 71 0B 51 09 41 C6 7B  Æ▌ÿ▌UþL▌_q.Q.AÆ{
00000976  DA 2C F3 FD B9 3A 19 28  DA 2C 98 23 91 DE 98 8C  Ú,óý¹:.(Ú,▌#´Þ▌▌
```



```
00001008  B4 1D 2B 9A 78 AF CA 5A  6D 90 65 71 A1 F1 81 28  ´.+▮x¯ÊZm.eqiñ.(
00001024  F0 E9 A4 57 D0 D8 2F 2C  14 71 48 2C F3 EA D8 18  ðé¤WÐØ/,.qH,óêØ.
00001040  F4 A4 46 15 7E 2C D5 BB  B7 7D 85 77 74 FF 2F 5A  ô¤F.~,Õ»·}▮wtÿ/Z
00001056  DB 70 C5 42 03 5A 8D 70  A1 C7 11 0F AE BF 37 2A  ÛpÅB.Z.piÇ..®¿7*
00001072  F6 CD C4 0F 0A 96 20 BE  FD C4 2B C1 01 C5 87 15  öÍÄ..▮ ¾ýÄ+Á.Å▮.
00001088  9D AB 1B 87 B5 26 A5 B4  25 5E B6 FA EA C3 6B 0F  .«.▮µ&¥´%^¶úêÃk.
00001104  FB 76 AA 9C 79 ED 51 8D  EB 87 08 D4 60 16 B3 FB  ûvª▮yíQ.ë▮.Ô`.³û
00001120  E7 27 AB B7 DC 83 42 3B  F7 C3 32 2A 15 53 81 C5  ç'«·Ü▮B;÷Ã2*.S.Å
00001136  29 D1 FF C7 D3 42 D4 80  0D 3F 58 24 D0 D7 9F E6  )ÑÿÇÓBÔ▮.?X$Ð×▮æ
```

Figure 2. Shellcode configuration structure.

A breakdown of the configuration structure shown in Figure 2 is below (from top to bottom):

- Highlighted in neon green are the two, 16-byte keys used for XORing values throughout the shellcode.
  7D 38 BA FD E1 C8 D2 DF B6 EE 33 F9 14 BF 52 96
  71 17 DF E4 AE 3B A9 F2 D5 3D 75 CC D3 0D 57 72
- Highlighted in light blue are the two bytes computed from the host's Quick Edit Registry key.
  E8 03
- Highlighted in orange are the four bytes that represent the shellcode version.
  30 2E 32 34 (0.24)
- Highlighted in pink are the 17 bytes that make up the C2 domain. Bitwise NOT (unsigned byte) to decode the values including the NULL.
  88 98 CE D1 96 91 94 9A 8C 93 96 89 9A D1 9C 90 92
- Highlighted in dark green are the 103 bytes that are used for pattern elimination. XOR with 0xFF to NULL values.
  FF FF FF FF FF FF FF FF FF FF FF…
- Highlighted in magenta are the two bytes that make up the target C2 port.
  BB 01
- Highlighted in light yellow are the resolved function pointers used by the shellcode.
  92 13 73 33 37 02
- Highlighted in dark cyan are the 112 bytes that make up the function pointer sizes used to encrypt or decrypt function blocks.
  EE 01
- Highlighted in dark red are the 289 bytes that make up the resolved Windows API functions used by the shellcode
  A0 2E 52 CC FC 7F 00 00…

Network Communications

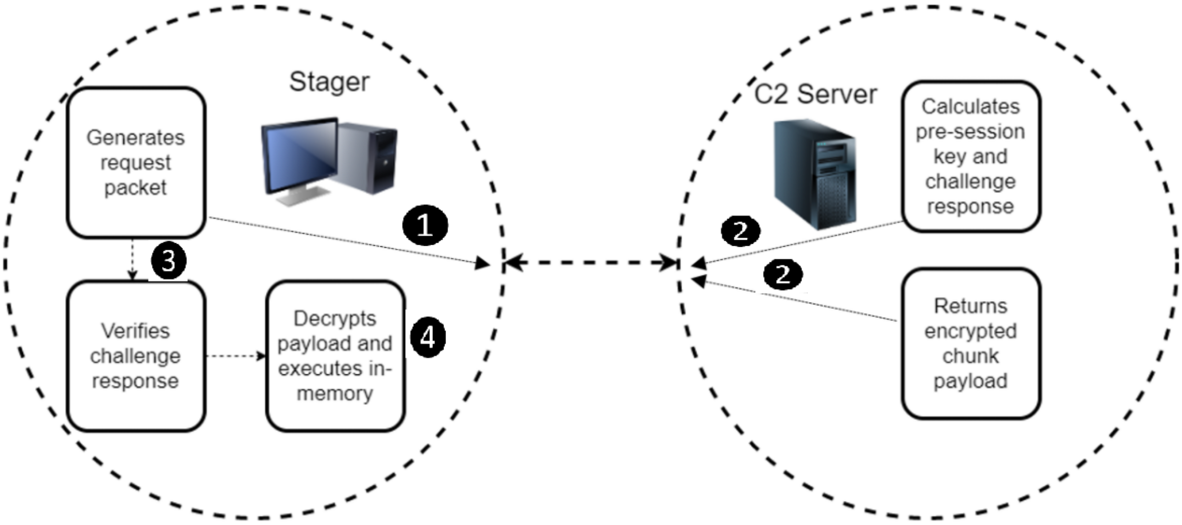

Stager
Generates request packet
1
3
Verifies challenge response
Decrypts payload and executes in-memory
4
C2 Server
Calculates pre-session key and challenge response
2
2
Returns encrypted chunk payload

Figure 3. Stager communication flow.

Before communicating with the C2 server, the shellcode flushes the host's DNS cache by performing the following:

1. Loads module dnsapi.dll
2. Calls API DnsFlushResolverCache

When this API is called, all domains resolved are cleared from the host's DNS cache, not just the target C2 server. This forces the host to resolve the current IP associated with the C2 domain, ensuring that communication continues as network infrastructure becomes compromised or unavailable. It also implies the developers own the domain and can update the IP.

The stager begins by computing 10 bytes of data to send to the C2 server. These 10 bytes make up a challenge request packet. The stager sends the challenge request to the C2 and waits for a challenge response. When received and properly decrypted, the stager checks for magic values or signature bytes at specific offsets. If this check fails, the network connection is aborted. This check ensures trusted communication with the intended C2 server and initiates the download of the payload.

I. Stager Generates Challenge Request Packet

The stager computes a 10-byte challenge request containing information for the C2, to include the size of the data (being the session keys) to be received next. The challenge request and session keys are sent to the C2 simultaneously. Example request:

26BCFCCE738A211F3763

II. C2 Server Decrypts Challenge Request Packet

The C2 decrypts the challenge request packet using the following steps:

1. First byte will be XORed with the second byte, second byte with third byte…until byte 10, followed by:

A. Byte 7 is updated from the result of (byte 7 XORed with byte 3).
B. Byte 2 is updated from the result of (byte 2 XORed with byte 0).
C. Byte 8 is updated from the result of (byte 8 XORed with byte 0).
D. Byte 9 is updated from the result of (byte 9 XORed with byte 5).

2. Final value is XORed with key 0x3FDA5F9AD85D50C77E6A

The challenge request decrypts to the following (represented as hex bytes):

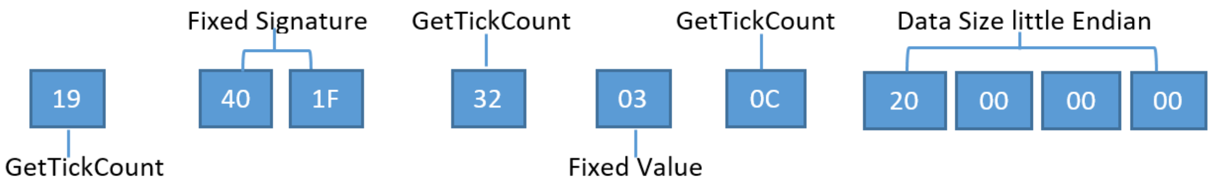


Figure 4. Decrypted request challenge.

The last four bytes of the decrypted request packet inform the C2 server of the size of the expected network traffic to follow. As shown above, the value is 0x20, or 32 bytes. These 32 bytes make up the session keys used by the C2 server to encrypt a server challenge response and encrypt the payload.

Example of session keys received by the C2 server:

Session key 1–> 8C931D4F764B0661C26D77239EB454CA

Session key 2–> 7A4DD0AA6C3F37CDBDAFA4CBD6B27697

The challenge request packet and session keys are computed for each beacon and therefore would always be unique.

III. C2 Authenticates With the Stager

The C2 uses the session keys to build the RC4 state box and as an XOR key for encryption and decryption.

**It should be noted that the use of session key 2 is not yet fully understood, and it did not appear to be used to communicate with the stager.*

1. The pre-session key is computed using session key 1 (first 16 bytes) as follows:
Pre-Session Key = session key 1 XOR
0X6162636465666768696A6B6C6D6E6F00

2. Using the computed pre-session key from step 1, the C2 server builds the RC4 Key Scheduling Algorithm (KSA). It is computed as follows:

a. Build the RC4 KSA using the following inputs to the below function:
data = 16-byte key 0x0C2F65194FF37B2D63D34635C7B205E4
key = 16-byte computed pre-session key from step 1

Example RC4 (modified) KSA routine:

```
def rc4_KSA(data, key):
    x = 0
    box = range(258)
    box[256] = 0
    box[257] = 0
    for i in range(256):
    x = (x + box[i] + ord(key[i % len(key)])) % 256
    box[i], box[x] = box[x], box[i]
    return box
```

**Note about the input parameter "data" for the KSA routine: It is the XOR result of the two 16-byte keys shown neon green in Figure 2. Shellcode Configuration Structure.*

3. Construct 10-byte server challenge response header using the hex values shown in Figure 5.

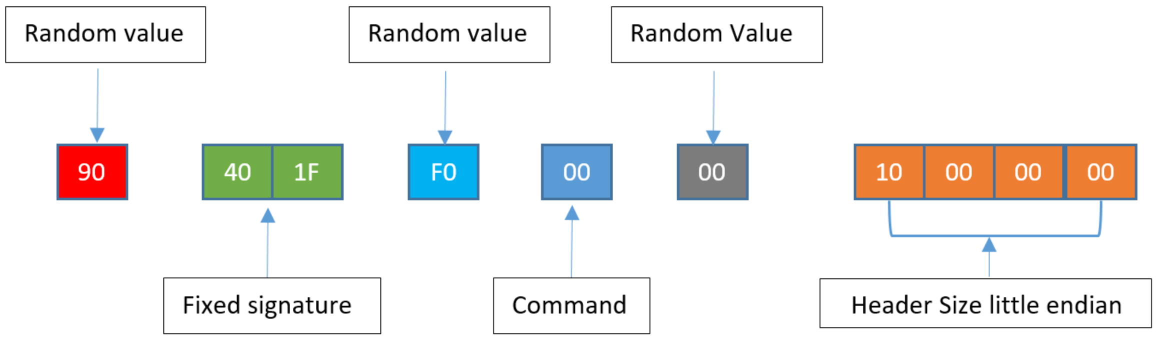


Figure 5. Server Command Challenge Header

4. Encrypt server challenge response header from step 3:

a. XOR 10-byte server challenge with key 0x33836E6B3FAA6AC464DA and perform the following:

i. Byte 7 is updated from the result of (byte 7 XORed with byte 3).
ii. Byte 2 is updated from the result of (byte 2 XORed with byte 0).
iii. Byte 8 is updated from the result of (byte 8 XORed with byte 0).
iv. Byte 9 is updated from the result of (byte 9 XORed with byte 5).

b. Encrypted server challenge response header = result of 4(a)

5. Compute final authentication key:

a. XOR the following values:

i. 0x0C2F65194FF37B2D63D34635C7B205E4
ii. Value computed from step 1, i.e. Pre-Session Key

**The 16-byte value in 5.a.i is the same input parameter used in the KSA algorithm in step 2. The stager expects this key from the C2 otherwise the session is aborted.*

The values generated in steps 4 and 5 make up the complete server challenge response. At this point, the C2 sends the server challenge response to the stager, completing the authentication process.

IV. C2 Encrypts and Transmits Payload

Next, the C2 prepares to send a command to the stager. BendyBear only supports one type of command: payload download.

1. Build a 10-byte command header using the hex values shown in Figure 6.

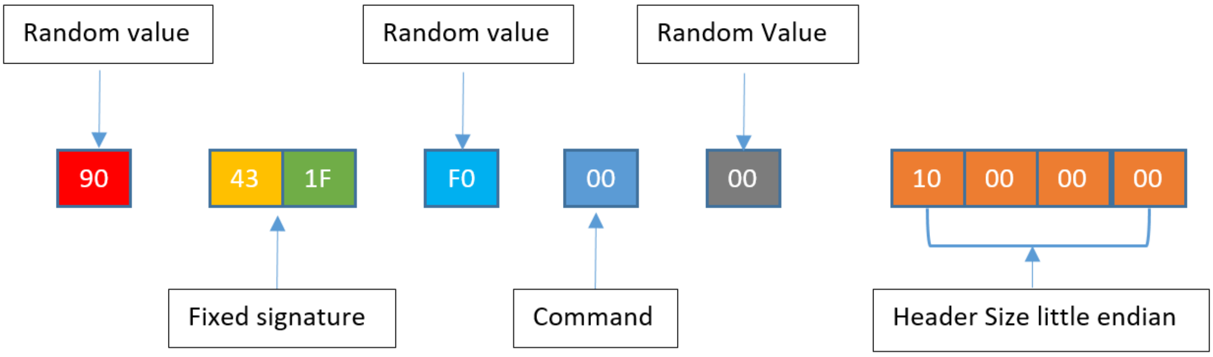


Figure 6. Updated server command challenge header.

The only change to the header is the fixed signature value from 0x40 to 0x43.

2. Encrypt the command header from step 1:

The following is an example of a RC4 modified routine that can be used. The first argument, box, would be the S-Box computed in step III.2 and the second argument, data, would be the command header from step 1.

```
def rc4_Mod_Crypt(box, data):
  x = box[256]
  y = box[257]
  c = 0
  out = []
  for char in data:
    x = (x + 1) % 256
    y = (y + box[x]) % 256
    box[x], box[y] = box[y], box[x]
    z = ( (box[x] + box[y] )&0xff ) % 256
    al = rol( box[z],4,8 )
    out.append( chr( ord( data[c] ) ^ al ) )
    box[z] = al
    c+=1
  box[256] = x
  box[257] = y
  return ”.join(out)
```

3. Obtain the size of the payload and encrypt that value using the same RC4 algorithm as in step 2. The size of the payload should be the total decrypted size of the payload.

4. Encrypt and send the payload to the stager in chunks:

a. Read 4,086 bytes from the payload. This is the maximum chunk size that the stager will accept.

b. Build a command header (step 1 above) and update the following fields:

i. Header size = size of payload chunk.

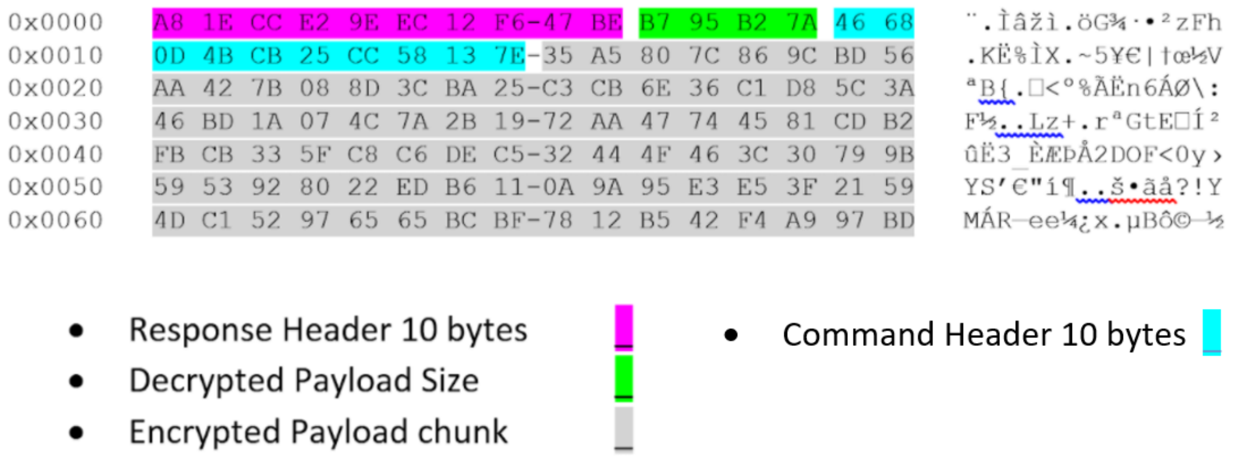
ii. Command = 1.

c. Send the updated 10-byte command header to the stager.

d. Send the encrypted payload chunk.

e. Repeat steps a – d until payload is sent.

Figure 7 shows an example of one payload chunk that is sent to the stager.



```
0x0000   A8 1E CC E2 9E EC 12 F6-47 BE B7 95 B2 7A 46 68   ¨.Ìâžì.öG¾·•²zFh
0x0010   0D 4B CB 25 CC 58 13 7E-35 A5 80 7C 86 9C BD 56   .KË%ÌX.~5¥€|†œ½V
0x0020   AA 42 7B 08 8D 3C BA 25-C3 CB 6E 36 C1 D8 5C 3A   ªB{.□<º%ÃËn6ÁØ\:
0x0030   46 BD 1A 07 4C 7A 2B 19-72 AA 47 74 45 81 CD B2   F½..Lz+.rªGtE□Í²
0x0040   FB CB 33 5F C8 C6 DE C5-32 44 4F 46 3C 30 79 9B   ûË3_ÈÆÞÅ2DOF<0y›
0x0050   59 53 92 80 22 ED B6 11-0A 9A 95 E3 E5 3F 21 59   YS’€"í¶..š•ãå?!Y
0x0060   4D C1 52 97 65 65 BC BF-78 12 B5 42 F4 A9 97 BD   MÁR—ee¼¿x.µBô©—½
```

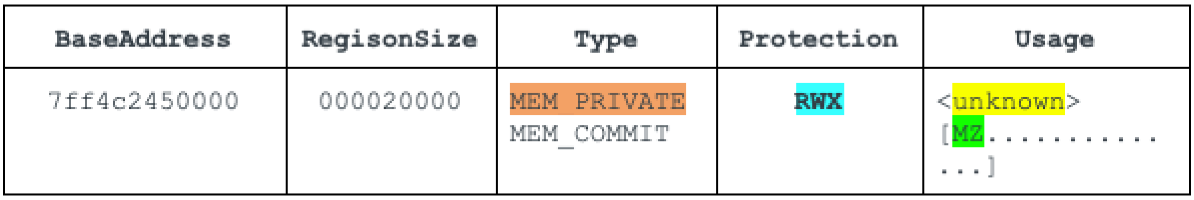
- Response Header 10 bytes 🟪
- Decrypted Payload Size 🟩
- Encrypted Payload chunk ⬜
- Command Header 10 bytes 🟦

Figure 7. Encrypted payload header and data.

Upon receiving each chunk, the stager strips the command header and decrypts the payload chunk in memory.

Payload In-Memory Loading

Once the payload is fully decrypted, the stager performs some basic checks to ensure that the payload conforms to a Windows executable. It validates the DOS and PE header and that the payload is a DLL. It then direct-memory loads the payload and calls into its entry point (AddressOfEntryPoint). The direct memory load of the payload emulates that of the Windows PE loader – LoadLibrary. As a result, the PEB LDR_DATA_TABLE_ENTRY metadata structures are not created and the PEB for the process running the shellcode has no record of the DLL loading, which can be used to detect rogue modules running on your host. This is visible in WinDbg running the command !address within the process that loaded the shellcode. An example is shown in Figure 8.



BaseAddress	RegisonSize	Type	Protection	Usage
7ff4c2450000	000020000	MEM_PRIVATE MEM_COMMIT	RWX	<unknown> [MZ..............]

Figure 8. Artifact of direct in-memory loaded DLL.

In-memory artifacts:

- Type is MEM_PRIVATE, meaning it is private to the process that loaded it. On Windows platforms, DLLs are typically loaded as MEM_IMAGE so that they can be shared between different processes to save memory space.
- Protection is PAGE_EXECUTE_READWRITE(RWX), which means the area is writable and executable with a memory area containing an MZ header. The MZ header is the in-memory loaded module.

The output of the WinDbg !address command shown in Figure 8 spots the anomalous entry. The memory address of module 0x7ff4c2450000 was the result of private memory allocation, protection set to RWX and usage containing an MZ header.

x64 Shellcode Behaviors

The following table describes the main behaviors of BendyBear.

Behavior Artifact	ATT&CK IDs
Query Registry HKEY_CURRENT_USER\Console\QuickEdit	T1012: Query Registry
Command and Control	T1573.002: Encrypted Channel: Asymmetric Cryptography
Payload transfer from remote host	T1105: Ingress Tool Transfer
Payloads in modified RC4-encrypted chunks	T1027.002: Obfuscated Files or Information: Software Packing
~65 calls to Windows API kernel32!GetTickCountKernel32 prior to the shellcode connecting to the C2 server. Used to encrypt or decrypt function blocks.	T1497.003: Time Based Evasion
Dynamic DLL Importing and API Lookups	T1106 Native API

52 iterations of the shellcode obtaining the process environment block (PEB) and checking for IsDebugger flag	T1082: System Information Discovery
Eight calls to msvcrt!time prior to connecting to the C2 server	
Clearing host’s DNS cache via API DNSAPI!DnsFlushResolverCache	
PEB _LDR_DATA_TABLE_ENTRY metadata structures are not created, and the PEB for the process running the shellcode has no record of the DLL loading.	
Loaded payload module (DLL) has a type of MEM_PRIVATE	

Table 1. x64 shellcode commands executed.

BendyBear vs. WaterBear

Attributes	WaterBear	BendyBear
File Type	EXE/DLL	Shellcode
Implant Type	Stage-2	Stage-0
Modified RC4	✓	✓
Additional Encryption	UNKNOWN	Extra XOR Computations
16-Byte XOR keys	✓	✓
Authenticated C2 Communications	✓	✓
Signature Verification Magic Bytes	1F 40 1F 43	1F 40 1F 43
Chunked Payloads	✓	✓
Polymorphic Code	✓	✓
In-Memory Loading	✓	✓
PEB Debugger Check	✓	✓
Pattern Elimination	✓	✓
Encrypt/Decrypt Function Routines	✓	✓
API Hooking	✓	✗

Process Hiding	✓	✗
Network Traffic Filtering	✓	✗

Table 2. Comparison of BendyBear and WaterBear.

File Type – WaterBear is a standalone PE/EXE. BendyBear is a x64 Shellcode that requires loader or code injection.

Implant Type – WaterBear is a stage-2 implant with many capabilities; BendyBear is a stage-0 downloader.

Modified RC4 Encryption – Both WaterBear and BendyBear use a modified RC4, but implement them slightly differently. WaterBear uses a 256 RC4 state box with byte shifting and addition within the key scheduling algorithm. BendyBear uses a 258 RC4 state box and performs XOR within the key scheduling algorithm.

Additional Encryption – While both use encryption as a way to conceal artifacts, BendyBear was found to contain additional XOR encryption steps.

16-Byte XOR Key – Both use the same 16-byte XOR key to create the pre-session key: 0x6162636465666768696A6B6C6D6E6f00

Authenticated C2 Communications – Both send an initial 10-byte challenge request followed by 32-byte session keys.

Signature Verification Magic Bytes – Both use the same matching magic byte verification values.

Chunked Payload – Both expect the payloads to be sent in encrypted chunks.

Polymorphic Code – Both employ code manipulation during runtime execution with random bytes.

In-Memory Loading – Both support the in-memory loading of payloads.

PEB Debugger Check – Both check to see if the code is being debugged.

Pattern Elimination –Both re-encrypt any decrypted strings upon use.

Encrypt/Decrypt Function Routines – Both WaterBear and BendyBear obfuscate runtime function addresses.

API Hooking – Variants of WaterBear implement API hooking, while BendyBear does not.

Process Hiding – Variants of WaterBear can hide processes via API hooking, while BendyBear does not support this capability.

Network Traffic Filtering – Variants of WaterBear can filter or hide network traffic via API hooking, while BendyBear does not support this capability.

Conclusion

The BendyBear shellcode contains advanced features that are not typically found in shellcode. The use of anti-analysis techniques and signature block verification indicate that the developers care about stealth and detection-evasion. Additionally, the use of custom cryptographic routines and byte manipulations suggest a high level of technical sophistication.

Palo Alto Networks customers can be protected from the attacks outlined in this blog in the following ways:

- The C2 domain used in this shellcode has been categorized as malware in DNS Security, URL Filtering and WildFire, which are security subscriptions for Next-Generation Firewall customers.
- Cortex XDR can identify and block the shellcode during execution.
- App-ID, the traffic classification system in Next-Generation Firewalls, is capable of identifying applications irrespective of port, protocol, encryption (SSH or SSL) or any other evasive tactic used by the application. This shellcode attempts to communicate over TCP port 443 with traffic that does not conform to proper SSL or any other known application. As a matter of best practice, we advise customers to block unknown outbound TCP traffic in their security policies.

Indicators of Compromise

Shellcode Samples

x64 – (version 0.24)
64CC899EC85F612270FCFB120A4C80D52D78E68B05CAF1014D2FE06522F1E2D0
wg1.inkeslive[.]com

x86 – (version 0.1)
49901034216a16cfd05c613f438eccee4a7bf6079a7988b3e7094d9498379558
web2008.rutentw[.]com

x86 WaterBear Loaders

The following executables have been identified as loaders/injectors that contain older WaterBear x86 shellcode. The shellcode code is identical to the x86 sample 49901034216…. (version 0.1) listed above.

5d1414b47d88e95ae6612d3fc211c29b35cc5db4a8a992f5e27cff5203ebf44b
9880ba4f93cade2f6bbb4cc8efdcf087e8ac51b5c209ee32ad8134eb87ef70e1
682122f34027e3f8025928d446989b02952449f5e5930c2670f8f789f41573ff
2a09ec2d6edadd06e18c841e0ed794ba3eeb21818476f75ccc0e5d40e08eac80
76ef704d21fbaaceca8a131429ccfb9f5de3d8f43a160ddd281ffeafc391eb98

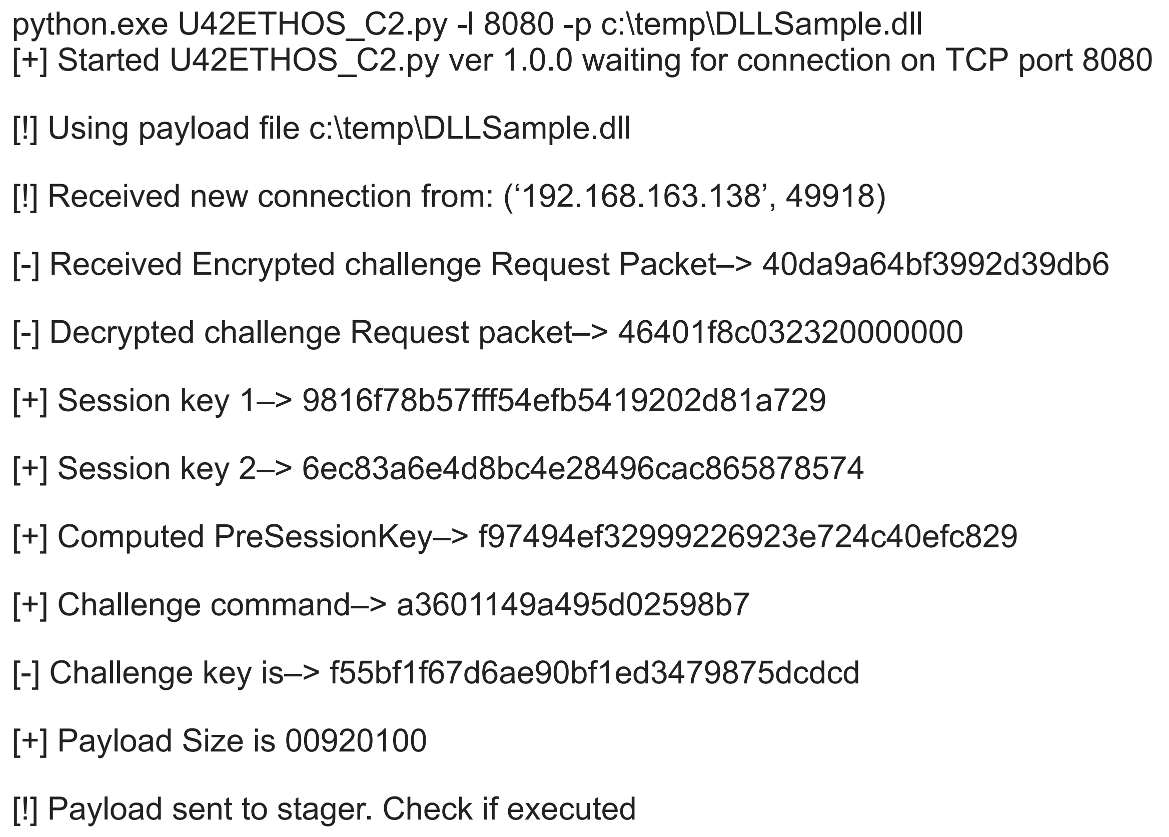
Additional Resources

Taiwan News – Taiwan urges blocking 11 China-linked phishing domains.
iThome News – The Bureau of Investigation's recent investigation of several cases of Taiwan Government agencies hacked.
TeamT5 – Evil Hidden in Shellcode: The Evolution of malware DbgPrint.
TrendMicro – WaterBear Returns, Uses API Hooking to Evade Security.
TrendMicro – The Trail of BlackTech's Cyber Espionage Campaigns.
CryCraft Technology Corp – Taiwan Government Targeted by Multiple Cyberattacks in April 2020 Part 1: Waterbear Malware
JPCERT/CC Eyes – ELF_PLEAD – Linux Malware Used by BlackTech

Appendix

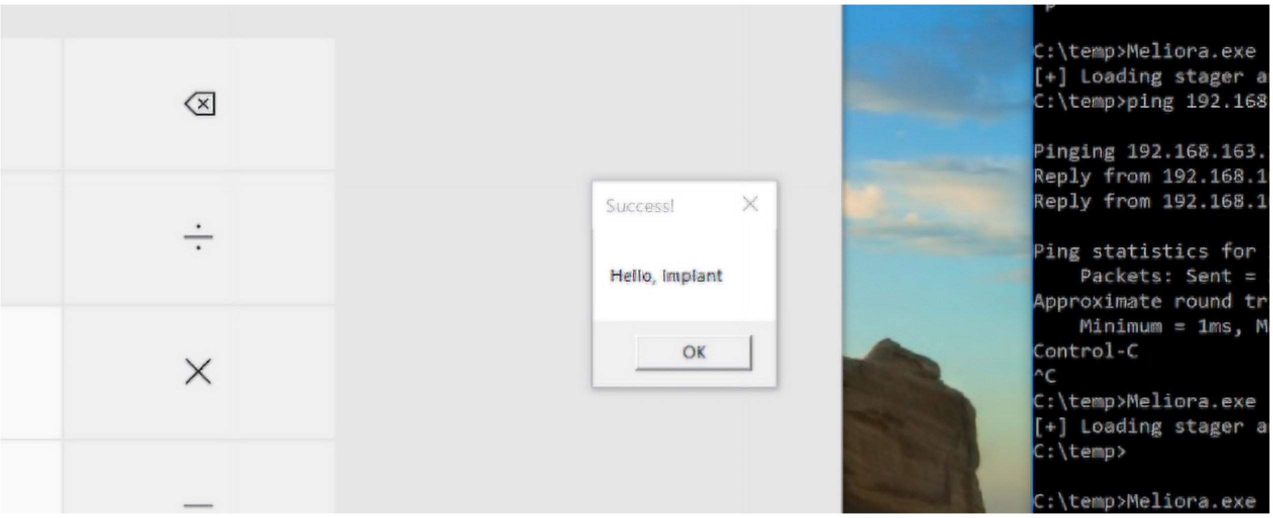
Shellcode Proof of Concept

Mock C2 server serving request to stager and sending a payload (DLL) that displays a message box:



```
python.exe U42ETHOS_C2.py -l 8080 -p c:\temp\DLLSample.dll
[+] Started U42ETHOS_C2.py ver 1.0.0 waiting for connection on TCP port 8080

[!] Using payload file c:\temp\DLLSample.dll

[!] Received new connection from: ('192.168.163.138', 49918)

[-] Received Encrypted challenge Request Packet–> 40da9a64bf3992d39db6

[-] Decrypted challenge Request packet–> 46401f8c032320000000

[+] Session key 1–> 9816f78b57fff54efb5419202d81a729

[+] Session key 2–> 6ec83a6e4d8bc4e28496cac865878574

[+] Computed PreSessionKey–> f97494ef32999226923e724c40efc829

[+] Challenge command–> a3601149a495d02598b7

[-] Challenge key is–> f55bf1f67d6ae90bf1ed3479875dcdcd

[+] Payload Size is 00920100

[!] Payload sent to stager. Check if executed
```

Figure 9. Unit 42 mock C2 server.



Success!

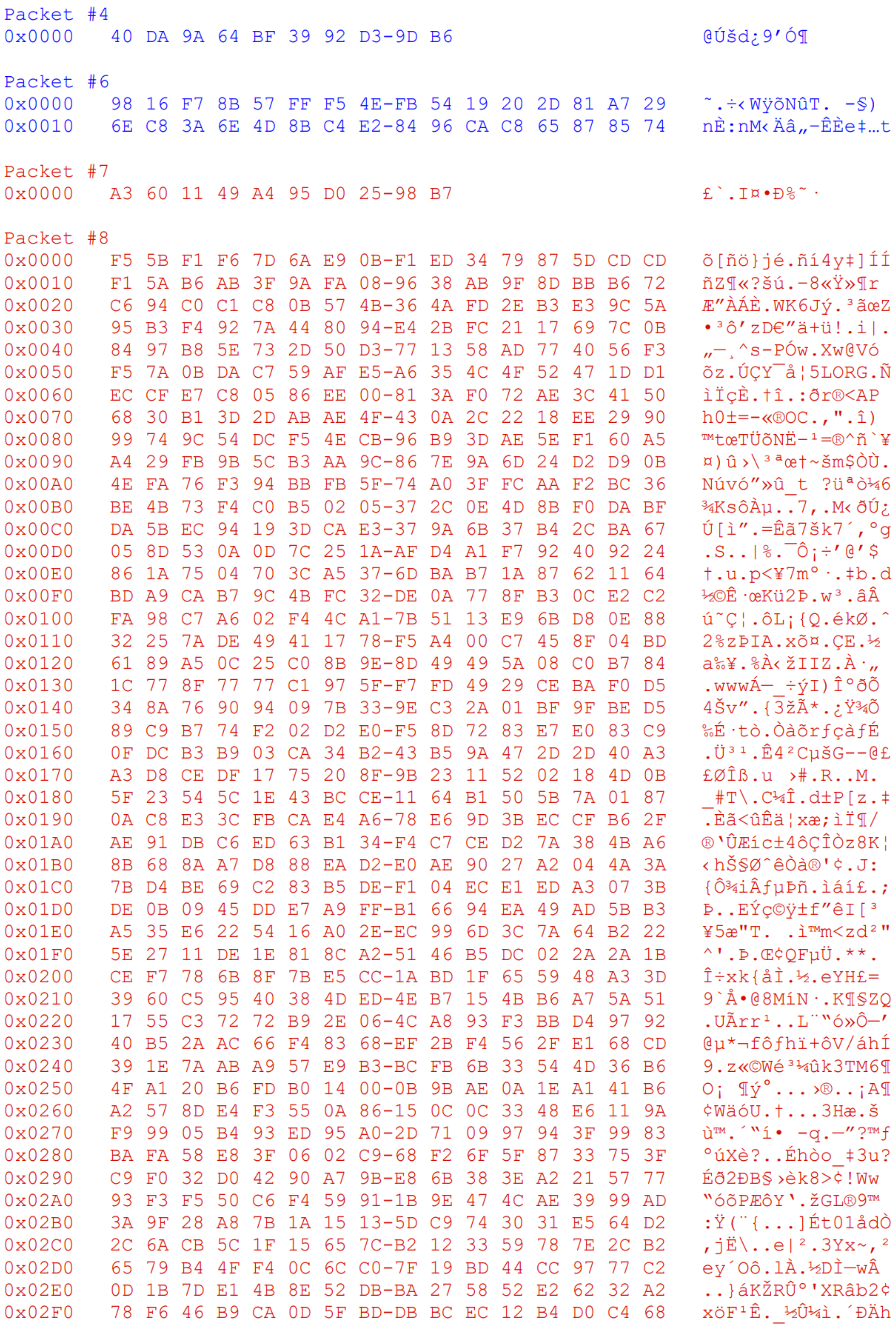
Hello, Implant

```
C:\temp>Meliora.exe
[+] Loading stager a
C:\temp>ping 192.168

Pinging 192.168.163.
Reply from 192.168.1
Reply from 192.168.1

Ping statistics for
    Packets: Sent =
Approximate round tr
    Minimum = 1ms, M
Control-C
^C
C:\temp>Meliora.exe
[+] Loading stager a
C:\temp>

C:\temp>Meliora.exe
```

Figure 10. Example stager in-memory loading test DLL

Figure 9 is a Python mock C2 server that was created by Unit 42 to interact with the stager. It is configured to listen on TCP port 8080, and the payload is a test DLL that launches calc.exe and displays a message box (Hello, Implant). Figure 10 is a Windows 10 host running the shellcode in memory via a custom loader. The shellcode was configured to communicate with the mock C2 server.

Network Traffic for the Above Payload (truncated):



```
Packet #4
0x0000   40 DA 9A 64 BF 39 92 D3-9D B6                      @Úšd¿9’Ó¶

Packet #6
0x0000   98 16 F7 8B 57 FF F5 4E-FB 54 19 20 2D 81 A7 29    ˜.÷‹WÿõNûT. -§)
0x0010   6E C8 3A 6E 4D 8B C4 E2-84 96 CA C8 65 87 85 74    nÈ:nM‹Äâ„–ÊÈe‡…t

Packet #7
0x0000   A3 60 11 49 A4 95 D0 25-98 B7                      £`.I¤•Ð%˜·

Packet #8
0x0000   F5 5B F1 F6 7D 6A E9 0B-F1 ED 34 79 87 5D CD CD    õ[ñö}jé.ñí4y‡]ÍÍ
0x0010   F1 5A B6 AB 3F 9A FA 08-96 38 AB 9F 8D BB B6 72    ñZ¶«?šú.–8«Ÿ»¶r
0x0020   C6 94 C0 C1 C8 0B 57 4B-36 4A FD 2E B3 E3 9C 5A    Æ”ÀÁÈ.WK6Jý.³ãœZ
0x0030   95 B3 F4 92 7A 44 80 94-E4 2B FC 21 17 69 7C 0B    •³ô’zD€”ä+ü!.i|.
0x0040   84 97 B8 5E 73 2D 50 D3-77 13 58 AD 77 40 56 F3    „—¸^s-PÓw.Xw@Vó
0x0050   F5 7A 0B DA C7 59 AF E5-A6 35 4C 4F 52 47 1D D1    õz.ÚÇY¯å¦5LORG.Ñ
0x0060   EC CF E7 C8 05 86 EE 00-81 3A F0 72 AE 3C 41 50    ìÏçÈ.†î.:ðr®<AP
0x0070   68 30 B1 3D 2D AB AE 4F-43 0A 2C 22 18 EE 29 90    h0±=-«®OC.,".î)
0x0080   99 74 9C 54 DC F5 4E CB-96 B9 3D AE 5E F1 60 A5    ™tœTÜõNË–¹=®^ñ`¥
0x0090   A4 29 FB 9B 5C B3 AA 9C-86 7E 9A 6D 24 D2 D9 0B    ¤)û›\³ªœ†~šm$ÒÙ.
0x00A0   4E FA 76 F3 94 BB FB 5F-74 A0 3F FC AA F2 BC 36    Núvó”»û_t ?üªò¼6
0x00B0   BE 4B 73 F4 C0 B5 02 05-37 2C 0E 4D 8B F0 DA BF    ¾KsôÀµ..7,.M‹ðÚ¿
0x00C0   DA 5B EC 94 19 3D CA E3-37 9A 6B 37 B4 2C BA 67    Ú[ì”.=Êã7šk7´,ºg
0x00D0   05 8D 53 0A 0D 7C 25 1A-AF D4 A1 F7 92 40 92 24    .S..|%.¯Ô¡÷’@’$
0x00E0   86 1A 75 04 70 3C A5 37-6D BA B7 1A 87 62 11 64    †.u.p<¥7mº·.‡b.d
0x00F0   BD A9 CA B7 9C 4B FC 32-DE 0A 77 8F B3 0C E2 C2    ½©Ê·œKü2Þ.w³.âÂ
0x0100   FA 98 C7 A6 02 F4 4C A1-7B 51 13 E9 6B D8 0E 88    ú˜Ç¦.ôL¡{Q.ékØ.ˆ
0x0110   32 25 7A DE 49 41 17 78-F5 A4 00 C7 45 8F 04 BD    2%zÞIA.xõ¤.ÇE.½
0x0120   61 89 A5 0C 25 C0 8B 9E-8D 49 49 5A 08 C0 B7 84    a‰¥.%À‹žIIZ.À·„
0x0130   1C 77 8F 77 77 C1 97 5F-F7 FD 49 29 CE BA F0 D5    .wwwÁ—_÷ýI)ÎºðÕ
0x0140   34 8A 76 90 94 09 7B 33-9E C3 2A 01 BF 9F BE D5    4Šv”.{3žÃ*.¿Ÿ¾Õ
0x0150   89 C9 B7 74 F2 02 D2 E0-F5 8D 72 83 E7 E0 83 C9    ‰É·tò.ÒàõrƒçàƒÉ
0x0160   0F DC B3 B9 03 CA 34 B2-43 B5 9A 47 2D 2D 40 A3    .Ü³¹.Ê4²CµšG--@£
0x0170   A3 D8 CE DF 17 75 20 8F-9B 23 11 52 02 18 4D 0B    £ØÎß.u ›#.R..M.
0x0180   5F 23 54 5C 1E 43 BC CE-11 64 B1 50 5B 7A 01 87    _#T\.C¼Î.d±P[z.‡
0x0190   0A C8 E3 3C FB CA E4 A6-78 E6 9D 3B EC CF B6 2F    .Èã<ûÊä¦xæ;ìÏ¶/
0x01A0   AE 91 DB C6 ED 63 B1 34-F4 C7 CE D2 7A 38 4B A6    ®‘ÛÆíc±4ôÇÎÒz8K¦
0x01B0   8B 68 8A A7 D8 88 EA D2-E0 AE 90 27 A2 04 4A 3A    ‹hŠ§ØˆêÒà®'¢.J:
0x01C0   7B D4 BE 69 C2 83 B5 DE-F1 04 EC E1 ED A3 07 3B    {Ô¾iÂƒµÞñ.ìáí£.;
0x01D0   DE 0B 09 45 DD E7 A9 FF-B1 66 94 EA 49 AD 5B B3    Þ..EÝç©ÿ±f”êI[³
0x01E0   A5 35 E6 22 54 16 A0 2E-EC 99 6D 3C 7A 64 B2 22    ¥5æ"T. .ì™m<zd²"
0x01F0   5E 27 11 DE 1E 81 8C A2-51 46 B5 DC 02 2A 2A 1B    ^'.Þ.Œ¢QFµÜ.**.
0x0200   CE F7 78 6B 8F 7B E5 CC-1A BD 1F 65 59 48 A3 3D    Î÷xk{åÌ.½.eYH£=
0x0210   39 60 C5 95 40 38 4D ED-4E B7 15 4B B6 A7 5A 51    9`Å•@8MíN·.K¶§ZQ
0x0220   17 55 C3 72 72 B9 2E 06-4C A8 93 F3 BB D4 97 92    .UÃrr¹..L¨“ó»Ô—’
0x0230   40 B5 2A AC 66 F4 83 68-EF 2B F4 56 2F E1 68 CD    @µ*¬fôƒhï+ôV/áhÍ
0x0240   39 1E 7A AB A9 57 E9 B3-BC FB 6B 33 54 4D 36 B6    9.z«©Wé³¼ûk3TM6¶
0x0250   4F A1 20 B6 FD B0 14 00-0B 9B AE 0A 1E A1 41 B6    O¡ ¶ý°...›®..¡A¶
0x0260   A2 57 8D E4 F3 55 0A 86-15 0C 0C 33 48 E6 11 9A    ¢WäóU.†...3Hæ.š
0x0270   F9 99 05 B4 93 ED 95 A0-2D 71 09 97 94 3F 99 83    ù™.´“í• -q.—”?™ƒ
0x0280   BA FA 58 E8 3F 06 02 C9-68 F2 6F 5F 87 33 75 3F    ºúXè?..Éhòo_‡3u?
0x0290   C9 F0 32 D0 42 90 A7 9B-E8 6B 38 3E A2 21 57 77    Éð2ÐB§›èk8>¢!Ww
0x02A0   93 F3 F5 50 C6 F4 59 91-1B 9E 47 4C AE 39 99 AD    “óõPÆôY‘.žGL®9™
0x02B0   3A 9F 28 A8 7B 1A 15 13-5D C9 74 30 31 E5 64 D2    :Ÿ(¨{...]Ét01ådÒ
0x02C0   2C 6A CB 5C 1F 15 65 7C-B2 12 33 59 78 7E 2C B2    ,jË\..e|².3Yx~,²
0x02D0   65 79 B4 4F F4 0C 6C C0-7F BD 44 CC 97 77 C2       ey´Oô.lÀ.½DÌ—wÂ
0x02E0   0D 1B 7D E1 4B 8E 52 DB-BA 27 58 52 E2 62 32 A2    ..}áKŽRÛº'XRâb2¢
0x02F0   78 F6 46 B9 CA 0D 5F BD-DB BC EC 12 B4 D0 C4 68    xöF¹Ê._½Û¼ì.´ÐÄh
```

Figure 11. Network traffic capture example.