CobaltStrikeParser/parse_beacon_config.py at master · Sentinel-One/CobaltStrikeParser · GitHub

github.com/Sentinel-One/CobaltStrikeParser/blob/master/parse_beacon_config.py

Sentinel-One

Sentinel-One/
CobaltStrikeParser

7 Contributors | 4 Issues | 656 Stars | 149 Forks

```
#!/usr/bin/python3
'''
Parses CobaltStrike Beacon's configuration from PE file or memory dump.
By Gal Kristal from SentinelOne (gkristal.w@gmail.com) @gal_kristal

Inspired by https://github.com/JPCERTCC/aa-tools/blob/master/cobaltstrikescan.py

TODO:
1. Parse headers modifiers
2. Dynamic size parsing
'''

from beacon_utils import *
from struct import unpack, unpack_from
```

```
from socket import inet_ntoa
from collections import OrderedDict
from netstruct import unpack as netunpack
import argparse
import io
import re
import pefile
import os
import hashlib
from io import BytesIO

THRESHOLD = 1100
COLUMN_WIDTH = 35
SUPPORTED_VERSIONS = (3, 4)
SILENT_CONFIGS = ['PublicKey', 'ProcInject_Stub', 'smbFrameHeader',
'tcpFrameHeader', 'SpawnTo']

def _cli_print(msg, end='\n'):
if __name__ == '__main__':
print(msg, end=end)

class confConsts:
MAX_SETTINGS = 64
TYPE_NONE = 0
TYPE_SHORT = 1
TYPE_INT = 2
TYPE_STR = 3

START_PATTERNS = {
```

```
3: b'\x69\x68\x69\x68\x69\x6b..\x69\x6b\x69\x68\x69\x6b..\x69\x6a',
4: b'\x2e\x2f\x2e\x2f\x2e\x2c..\x2e\x2c\x2e\x2f\x2e\x2c..\x2e'
}
START_PATTERN_DECODED = b'\x00\x01\x00\x01\x00\x02..\x00\x02\x00\x01\x00\x02..\x00'
CONFIG_SIZE = 4096
XORBYTES = {
3: 0x69,
4: 0x2e
}

class packedSetting:

def __init__(self, pos, datatype, length=0, isBlob=False, isHeaders=False, isIpAddress=False, isBool=False, isDate=False, boolFalseValue=0, isProcInjectTransform=False, isMalleableStream=False, hashBlob=False, enum=None, mask=None):
self.pos = pos
self.datatype = datatype
self.is_blob = isBlob
self.is_headers = isHeaders
self.is_ipaddress = isIpAddress
self.is_bool = isBool
self.is_date = isDate
self.is_malleable_stream = isMalleableStream
self.bool_false_value = boolFalseValue
self.is_transform = isProcInjectTransform
self.hashBlob = hashBlob
self.enum = enum
```

```
        self.mask = mask
        self.transform_get = None
        self.transform_post = None
        if datatype == confConsts.TYPE_STR and length == 0:
            raise(Exception("if datatype is TYPE_STR then length must not be 0"))

        self.length = length
        if datatype == confConsts.TYPE_SHORT:
            self.length = 2
        elif datatype == confConsts.TYPE_INT:
            self.length = 4


    def binary_repr(self):
        """
        Param number - Type - Length - Value
        """
        self_repr = bytearray(6)
        self_repr[1] = self.pos
        self_repr[3] = self.datatype
        self_repr[4:6] = self.length.to_bytes(2, 'big')
        return self_repr

    def parse_transformdata(self, data):
        '''
        Args:
            data (bytes): Raw communication transforam data

        Returns:
```

```
    dict: Dict of transform commands that should be convenient for communication forging

    '''
    dio = io.BytesIO(data)
    trans = {'ConstHeaders':[], 'ConstParams': [], 'Metadata': [], 'SessionId': [], 'Output': []}
    current_category = 'Constants'

    # TODO: replace all magic numbers here with enum
    while True:
        tstep = read_dword_be(dio)
        if tstep == 7:
            name = read_dword_be(dio)
            if self.pos == 12: # GET
                current_category = 'Metadata'
            else: # POST
                current_category = 'SessionId' if name == 0 else 'Output'
        elif tstep in (1, 2, 5, 6):
            length = read_dword_be(dio)
            step_data = dio.read(length).decode()
            trans[current_category].append(BeaconSettings.TSTEPS[tstep] + ' "' + step_data + '"')
        elif tstep in (10, 16, 9):
            length = read_dword_be(dio)
            step_data = dio.read(length).decode()
            if tstep == 9:
                trans['ConstParams'].append(step_data)
            else:
                trans['ConstHeaders'].append(step_data)
```

```
    elif tstep in (3, 4, 13, 8, 11, 12, 15):
    trans[current_category].append(BeaconSettings.TSTEPS[tstep])
    else:
    break

    if self.pos == 12:
    self.transform_get = trans
    else:
    self.transform_post = trans

    return trans


    def pretty_repr(self, full_config_data):
    data_offset = full_config_data.find(self.binary_repr())
    if data_offset < 0 and self.datatype == confConsts.TYPE_STR:
    self.length = 16
    while self.length < 2048:
    data_offset = full_config_data.find(self.binary_repr())
    if data_offset > 0:
    break
    self.length *= 2

    if data_offset < 0:
    return 'Not Found'

    repr_len = len(self.binary_repr())
    conf_data = full_config_data[data_offset + repr_len : data_offset + repr_len + self.length]
    if self.datatype == confConsts.TYPE_SHORT:
    conf_data = unpack('>H', conf_data)[0]
```

```
if self.is_bool:
ret = 'False' if conf_data == self.bool_false_value else 'True'
return ret
elif self.enum:
return self.enum[conf_data]
elif self.mask:
ret_arr = []
for k,v in self.mask.items():
if k == 0 and k == conf_data:
ret_arr.append(v)
if k & conf_data:
ret_arr.append(v)
return ret_arr
else:
return conf_data

elif self.datatype == confConsts.TYPE_INT:
if self.is_ipaddress:
return inet_ntoa(conf_data)

else:
conf_data = unpack('>i', conf_data)[0]
if self.is_date and conf_data != 0:
fulldate = str(conf_data)
return "%s-%s-%s" % (fulldate[0:4], fulldate[4:6], fulldate[6:])

return conf_data

if self.is_blob:
```

```
if self.enum != None:
ret_arr = []
i = 0
while i < len(conf_data):
v = conf_data[i]
if v == 0:
return ret_arr
v = self.enum[v]
if v:
ret_arr.append(v)
i+=1

# Only EXECUTE_TYPE for now
else:
# Skipping unknown short value in the start
string1 = netunpack(b'I$', conf_data[i+3:])[0].decode()
string2 = netunpack(b'I$', conf_data[i+3+4+len(string1):])[0].decode()
ret_arr.append("%s:%s" % (string1.strip('\x00'),string2.strip('\x00')))
i += len(string1) + len(string2) + 11


if self.is_transform:
if conf_data == bytes(len(conf_data)):
return 'Empty'

ret_arr = []
prepend_length = unpack('>I', conf_data[0:4])[0]
prepend = conf_data[4 : 4+prepend_length]
append_length_offset = prepend_length + 4
```

```
append_length = unpack('>I', conf_data[append_length_offset :
append_length_offset+4])[0]
append = conf_data[append_length_offset+4 : append_length_offset+4+append_length]
ret_arr.append(prepend)
ret_arr.append(append if append_length < 256 and append != bytes(append_length)
else 'Empty')
return ret_arr

if self.is_malleable_stream:
prog = []
fh = io.BytesIO(conf_data)
while True:
op = read_dword_be(fh)
if not op:
break
if op == 1:
l = read_dword_be(fh)
prog.append("Remove %d bytes from the end" % l)
elif op == 2:
l = read_dword_be(fh)
prog.append("Remove %d bytes from the beginning" % l)
elif op == 3:
prog.append("Base64 decode")
elif op == 8:
prog.append("NetBIOS decode 'a'")
elif op == 11:
prog.append("NetBIOS decode 'A'")
elif op == 13:
```

```
prog.append("Base64 URL-safe decode")
elif op == 15:
prog.append("XOR mask w/ random key")

conf_data = prog
if self.hashBlob:
conf_data = conf_data.strip(b'\x00')
conf_data = hashlib.md5(conf_data).hexdigest()

return conf_data

if self.is_headers:
return self.parse_transformdata(conf_data)

conf_data = conf_data.strip(b'\x00').decode()
return conf_data


class BeaconSettings:

BEACON_TYPE = {0x0: "HTTP", 0x1: "Hybrid HTTP DNS", 0x2: "SMB", 0x4: "TCP", 0x8: "HTTPS", 0x10: "Bind TCP"}
ACCESS_TYPE = {0x0: "Use proxy server (manual)", 0x1: "Use direct connection", 0x2: "Use IE settings", 0x4: "Use proxy server (credentials)"}
EXECUTE_TYPE = {0x1: "CreateThread", 0x2: "SetThreadContext", 0x3: "CreateRemoteThread", 0x4: "RtlCreateUserThread", 0x5: "NtQueueApcThread", 0x6: None, 0x7: None, 0x8: "NtQueueApcThread-s"}
ALLOCATION_FUNCTIONS = {0: "VirtualAllocEx", 1: "NtMapViewOfSection"}
TSTEPS = {1: "append", 2: "prepend", 3: "base64", 4: "print", 5: "parameter", 6: "header", 7: "build", 8: "netbios", 9: "const_parameter", 10: "const_header", 11: "netbiosu", 12: "uri_append", 13: "base64url", 14: "strrep", 15: "mask", 16: "const_host_header"}
```

```
ROTATE_STRATEGY = ["round-robin", "random", "failover", "failover-5x", "failover-50x", "failover-100x", "failover-1m", "failover-5m", "failover-15m", "failover-30m", "failover-1h", "failover-3h", "failover-6h", "failover-12h", "failover-1d", "rotate-1m", "rotate-5m", "rotate-15m", "rotate-30m", "rotate-1h", "rotate-3h", "rotate-6h", "rotate-12h", "rotate-1d" ]

def __init__(self, version):
if version not in SUPPORTED_VERSIONS:
_cli_print("Error: Only supports version 3 and 4, not %d" % version)
return
self.version = version
self.settings = OrderedDict()
self.init()

def init(self):
self.settings['BeaconType'] = packedSetting(1, confConsts.TYPE_SHORT, mask=self.BEACON_TYPE)
self.settings['Port'] = packedSetting(2, confConsts.TYPE_SHORT)
self.settings['SleepTime'] = packedSetting(3, confConsts.TYPE_INT)
self.settings['MaxGetSize'] = packedSetting(4, confConsts.TYPE_INT)
self.settings['Jitter'] = packedSetting(5, confConsts.TYPE_SHORT)
self.settings['MaxDNS'] = packedSetting(6, confConsts.TYPE_SHORT)
# Silenced config
self.settings['PublicKey'] = packedSetting(7, confConsts.TYPE_STR, 256, isBlob=True)
self.settings['PublicKey_MD5'] = packedSetting(7, confConsts.TYPE_STR, 256, isBlob=True, hashBlob=True)
self.settings['C2Server'] = packedSetting(8, confConsts.TYPE_STR, 256)
self.settings['UserAgent'] = packedSetting(9, confConsts.TYPE_STR, 128)
# TODO: Concat with C2Server?
self.settings['HttpPostUri'] = packedSetting(10, confConsts.TYPE_STR, 64)
```

```
# This is how the server transforms its communication to the beacon
# ref: https://www.cobaltstrike.com/help-malleable-c2 | https://usualsuspect.re/article/cobalt-strikes-malleable-c2-under-the-hood
# TODO: Switch to isHeaders parser logic
self.settings['Malleable_C2_Instructions'] = packedSetting(11, confConsts.TYPE_STR, 256, isBlob=True,isMalleableStream=True)
# This is the way the beacon transforms its communication to the server
# TODO: Change name to HttpGet_Client and HttpPost_Client
self.settings['HttpGet_Metadata'] = packedSetting(12, confConsts.TYPE_STR, 256, isHeaders=True)
self.settings['HttpPost_Metadata'] = packedSetting(13, confConsts.TYPE_STR, 256, isHeaders=True)

self.settings['SpawnTo'] = packedSetting(14, confConsts.TYPE_STR, 16, isBlob=True)
self.settings['PipeName'] = packedSetting(15, confConsts.TYPE_STR, 128)
# Options 16-18 are deprecated in 3.4
self.settings['DNS_Idle'] = packedSetting(19, confConsts.TYPE_INT, isIpAddress=True)
self.settings['DNS_Sleep'] = packedSetting(20, confConsts.TYPE_INT)
# Options 21-25 are for SSHAgent
self.settings['SSH_Host'] = packedSetting(21, confConsts.TYPE_STR, 256)
self.settings['SSH_Port'] = packedSetting(22, confConsts.TYPE_SHORT)
self.settings['SSH_Username'] = packedSetting(23, confConsts.TYPE_STR, 128)
self.settings['SSH_Password_Plaintext'] = packedSetting(24, confConsts.TYPE_STR, 128)
self.settings['SSH_Password_Pubkey'] = packedSetting(25, confConsts.TYPE_STR, 6144)
self.settings['SSH_Banner'] = packedSetting(54, confConsts.TYPE_STR, 128)

self.settings['HttpGet_Verb'] = packedSetting(26, confConsts.TYPE_STR, 16)
self.settings['HttpPost_Verb'] = packedSetting(27, confConsts.TYPE_STR, 16)
```

```
self.settings['HttpPostChunk'] = packedSetting(28, confConsts.TYPE_INT)
self.settings['Spawnto_x86'] = packedSetting(29, confConsts.TYPE_STR, 64)
self.settings['Spawnto_x64'] = packedSetting(30, confConsts.TYPE_STR, 64)
# Whether the beacon encrypts his communication, should be always on (1) in beacon 4
self.settings['CryptoScheme'] = packedSetting(31, confConsts.TYPE_SHORT)
self.settings['Proxy_Config'] = packedSetting(32, confConsts.TYPE_STR, 128)
self.settings['Proxy_User'] = packedSetting(33, confConsts.TYPE_STR, 64)
self.settings['Proxy_Password'] = packedSetting(34, confConsts.TYPE_STR, 64)
self.settings['Proxy_Behavior'] = packedSetting(35, confConsts.TYPE_SHORT,
enum=self.ACCESS_TYPE)
# Option 36 is deprecated in beacon < 4.5
self.settings['Watermark_Hash'] = packedSetting(36, confConsts.TYPE_STR, 32)
self.settings['Watermark'] = packedSetting(37, confConsts.TYPE_INT)
self.settings['bStageCleanup'] = packedSetting(38, confConsts.TYPE_SHORT,
isBool=True)
self.settings['bCFGCaution'] = packedSetting(39, confConsts.TYPE_SHORT,
isBool=True)
self.settings['KillDate'] = packedSetting(40, confConsts.TYPE_INT, isDate=True)
# Inner parameter, does not seem interesting so silencing
#self.settings['textSectionEnd (0 if !sleep_mask)'] = packedSetting(41,
confConsts.TYPE_INT)

#TODO: dynamic size parsing
#self.settings['ObfuscateSectionsInfo'] = packedSetting(42, confConsts.TYPE_STR, %d,
isBlob=True)
self.settings['bProcInject_StartRWX'] = packedSetting(43, confConsts.TYPE_SHORT,
isBool=True, boolFalseValue=4)
self.settings['bProcInject_UseRWX'] = packedSetting(44, confConsts.TYPE_SHORT,
isBool=True, boolFalseValue=32)
self.settings['bProcInject_MinAllocSize'] = packedSetting(45, confConsts.TYPE_INT)
```

```
self.settings['ProcInject_PrependAppend_x86'] = packedSetting(46, confConsts.TYPE_STR, 256, isBlob=True, isProcInjectTransform=True)

self.settings['ProcInject_PrependAppend_x64'] = packedSetting(47, confConsts.TYPE_STR, 256, isBlob=True, isProcInjectTransform=True)

self.settings['ProcInject_Execute'] = packedSetting(51, confConsts.TYPE_STR, 128, isBlob=True, enum=self.EXECUTE_TYPE)

# If True then allocation is using NtMapViewOfSection

self.settings['ProcInject_AllocationMethod'] = packedSetting(52, confConsts.TYPE_SHORT, enum=self.ALLOCATION_FUNCTIONS)


# Unknown data, silenced for now

self.settings['ProcInject_Stub'] = packedSetting(53, confConsts.TYPE_STR, 16, isBlob=True)

self.settings['bUsesCookies'] = packedSetting(50, confConsts.TYPE_SHORT, isBool=True)

self.settings['HostHeader'] = packedSetting(54, confConsts.TYPE_STR, 128)


# Silenced as I've yet to test it on a sample with those options

self.settings['smbFrameHeader'] = packedSetting(57, confConsts.TYPE_STR, 128, isBlob=True)

self.settings['tcpFrameHeader'] = packedSetting(58, confConsts.TYPE_STR, 128, isBlob=True)

self.settings['headersToRemove'] = packedSetting(59, confConsts.TYPE_STR, 64)


# DNS Beacon

self.settings['DNS_Beaconing'] = packedSetting(60, confConsts.TYPE_STR, 33)

self.settings['DNS_get_TypeA'] = packedSetting(61, confConsts.TYPE_STR, 33)

self.settings['DNS_get_TypeAAAA'] = packedSetting(62, confConsts.TYPE_STR, 33)

self.settings['DNS_get_TypeTXT'] = packedSetting(63, confConsts.TYPE_STR, 33)

self.settings['DNS_put_metadata'] = packedSetting(64, confConsts.TYPE_STR, 33)

self.settings['DNS_put_output'] = packedSetting(65, confConsts.TYPE_STR, 33)

self.settings['DNS_resolver'] = packedSetting(66, confConsts.TYPE_STR, 15)
```

```
self.settings['DNS_strategy'] = packedSetting(67, confConsts.TYPE_SHORT,
enum=self.ROTATE_STRATEGY)
self.settings['DNS_strategy_rotate_seconds'] = packedSetting(68,
confConsts.TYPE_INT)
self.settings['DNS_strategy_fail_x'] = packedSetting(69, confConsts.TYPE_INT)
self.settings['DNS_strategy_fail_seconds'] = packedSetting(70, confConsts.TYPE_INT)

# Retry settings (CS 4.5+ only)
self.settings['Retry_Max_Attempts'] = packedSetting(71, confConsts.TYPE_INT)
self.settings['Retry_Increase_Attempts'] = packedSetting(72, confConsts.TYPE_INT)
self.settings['Retry_Duration'] = packedSetting(73, confConsts.TYPE_INT)


class cobaltstrikeConfig:
def __init__(self, f):
'''
f: file path or file-like object
'''
self.data = None
if isinstance(f, str):
with open(f, 'rb') as fobj:
self.data = fobj.read()
else:
self.data = f.read()

"""Parse the CobaltStrike configuration"""

@staticmethod
def decode_config(cfg_blob, version):
return bytes([cfg_offset ^ confConsts.XORBYTES[version] for cfg_offset in cfg_blob])
```

```
def _parse_config(self, version, quiet=False, as_json=False):
'''
Parses beacon's configuration from beacon PE or memory dump.
Returns json of config is found; else it returns None.

:int version: Try a specific version (3 or 4), or leave None to try both of them
:bool quiet: Whether to print missing or empty settings
:bool as_json: Whether to dump as json
'''
re_start_match = re.search(confConsts.START_PATTERNS[version], self.data)
re_start_decoded_match = re.search(confConsts.START_PATTERN_DECODED,
self.data)

if not re_start_match and not re_start_decoded_match:
return None
encoded_config_offset = re_start_match.start() if re_start_match else -1
decoded_config_offset = re_start_decoded_match.start() if re_start_decoded_match
else -1

if encoded_config_offset >= 0:
full_config_data = cobaltstrikeConfig.decode_config(self.data[encoded_config_offset :
encoded_config_offset + confConsts.CONFIG_SIZE], version=version)
else:
full_config_data = self.data[decoded_config_offset : decoded_config_offset +
confConsts.CONFIG_SIZE]

parsed_config = {}
settings = BeaconSettings(version).settings.items()
for conf_name, packed_conf in settings:
parsed_setting = packed_conf.pretty_repr(full_config_data)
```

```
parsed_config[conf_name] = parsed_setting
if as_json:
continue

if conf_name in SILENT_CONFIGS:
continue

if parsed_setting == 'Not Found' and quiet:
continue

conf_type = type(parsed_setting)
if conf_type in (str, int, bytes):
if quiet and conf_type == str and parsed_setting.strip() == '':
continue
_cli_print("{: <{width}} - {val}".format(conf_name, width=COLUMN_WIDTH-3,
val=parsed_setting))

elif parsed_setting == []:
if quiet:
continue
_cli_print("{: <{width}} - {val}".format(conf_name, width=COLUMN_WIDTH-3,
val='Empty'))

elif conf_type == dict: # the beautifulest code
conf_data = []
for k in parsed_setting.keys():
if parsed_setting[k]:
conf_data.append(k)
for v in parsed_setting[k]:
conf_data.append('\t' + v)
```

```
    if not conf_data:
        continue
    _cli_print("{: <{width}} - {val}".format(conf_name, width=COLUMN_WIDTH-3, val=conf_data[0]))
    for val in conf_data[1:]:
        _cli_print(' ' * COLUMN_WIDTH, end='')
        _cli_print(val)

    elif conf_type == list: # list
        _cli_print("{: <{width}} - {val}".format(conf_name, width=COLUMN_WIDTH-3, val=parsed_setting[0]))
        for val in parsed_setting[1:]:
            _cli_print(' ' * COLUMN_WIDTH, end='')
            _cli_print(val)

    if as_json:
        _cli_print(json.dumps(parsed_config, cls=Base64Encoder))

    return parsed_config

def parse_config(self, version=None, quiet=False, as_json=False):
    '''
    Parses beacon's configuration from beacon PE or memory dump
    Returns json of config is found; else it returns None.

    :int version: Try a specific version (3 or 4), or leave None to try both of them
    :bool quiet: Whether to print missing or empty settings
    :bool as_json: Whether to dump as json
    '''

    if not version:
```

```
    for ver in SUPPORTED_VERSIONS:
    parsed = self._parse_config(version=ver, quiet=quiet, as_json=as_json)
    if parsed:
    return parsed
    else:
    return self._parse_config(version=version, quiet=quiet, as_json=as_json)
    return None


    def parse_encrypted_config_non_pe(self, version=None, quiet=False, as_json=False):
    self.data = decrypt_beacon(self.data)
    return self.parse_config(version=version, quiet=quiet, as_json=as_json)

    def parse_encrypted_config(self, version=None, quiet=False, as_json=False):
    '''
    Parses beacon's configuration from stager dll or memory dump
    Returns json of config is found; else it returns None.

    :bool quiet: Whether to print missing settings
    :bool as_json: Whether to dump as json
    '''

    try:
    pe = pefile.PE(data=self.data)
    except pefile.PEFormatError:
    return self.parse_encrypted_config_non_pe(version=version, quiet=quiet,
    as_json=as_json)

    data_sections = [s for s in pe.sections if s.Name.find(b'.data') != -1]
    if not data_sections:
```

```
    _cli_print("Failed to find .data section")
    return False
    data = data_sections[0].get_data()

    offset = 0
    key_found = False
    while offset < len(data):
    key = data[offset:offset+4]
    if key != bytes(4):
    if data.count(key) >= THRESHOLD:
    key_found = True
    size = int.from_bytes(data[offset-4:offset], 'little')
    encrypted_data_offset = offset+16 - (offset % 16)
    break

    offset += 4

    if not key_found:
    return False

    # decrypt
    enc_data = data[encrypted_data_offset:encrypted_data_offset+size]
    dec_data = []
    for i,c in enumerate(enc_data):
    dec_data.append(c ^ key[i % 4])

    dec_data = bytes(dec_data)
    self.data = dec_data
    return self.parse_config(version=version, quiet=quiet, as_json=as_json)
```

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Parses CobaltStrike Beacon's configuration from PE, memory dump or URL.")
    parser.add_argument("beacon", help="This can be a file path or a url (if started with http/s)")
    parser.add_argument("--json", help="Print as json", action="store_true", default=False)
    parser.add_argument("--quiet", help="Do not print missing or empty settings", action="store_true", default=False)
    parser.add_argument("--version", help="Try as specific cobalt version (3 or 4). If not specified, tries both.", type=int)
    args = parser.parse_args()

    if os.path.isfile(args.beacon):
        if cobaltstrikeConfig(args.beacon).parse_config(version=args.version, quiet=args.quiet, as_json=args.json) or \
            cobaltstrikeConfig(args.beacon).parse_encrypted_config(version=args.version, quiet=args.quiet, as_json=args.json):
            exit(0)

    elif args.beacon.lower().startswith('http'):
        x86_beacon_data = get_beacon_data(args.beacon, 'x86')
        x64_beacon_data = get_beacon_data(args.beacon, 'x64')
        if not x86_beacon_data and not x64_beacon_data:
            print("[-] Failed to find any beacon configuration")
            exit(1)

        conf_data = x86_beacon_data or x64_beacon_data
        if cobaltstrikeConfig(BytesIO(conf_data)).parse_config(version=args.version, quiet=args.quiet, as_json=args.json) or \
            cobaltstrikeConfig(BytesIO(conf_data)).parse_encrypted_config(version=args.version, quiet=args.quiet, as_json=args.json):
```

```
    exit(0)

else:
    print("[-] Target path is not an existing file or a C2 URL")
    exit(1)

print("[-] Failed to find any beacon configuration")
exit(1)
```