CAPEv2/Nighthawk.py at master · kevoreilly/CAPEv2 · GitHub

github.com/kevoreilly/CAPEv2/blob/master/modules/processing/parsers/CAPE/Nighthawk.py

kevoreilly

kevoreilly/**CAPEv2**

Malware Configuration And Payload Extraction

84 Contributors

19 Issues

1k Stars

249 Forks

```
import contextlib
import gzip
import itertools
import json
import struct

import pefile
import regex as re
from Crypto.Cipher import AES

DESCRIPTION = "NightHawk C2 DLL configuration parser."
AUTHOR = "Nikhil Ashok Hegde <@ka1do9>"


```

```
def _decode_str(encoded_string, plaintext_alphabet, ciphertext_alphabet):
"""
This function implements the substitution cipher that Nighthawk uses.
Encoded strings are decoded.
Borrowed from https://www.proofpoint.com/us/blog/threat-insight/nighthawk-and-coming-pentest-tool-likely-gain-threat-actor-notice
which is no longer available, but here's an archive link:
https://web.archive.org/web/20221128090619/https://www.proofpoint.com/us/blog/threat-insight/nighthawk-and-coming-pentest-tool-likely-gain-threat-actor-notice

:param encoded_string: String encoded with Nighthawk substitution cipher
:type encoded_string: <class 'bytes'>
:param plaintext_alphabet: Plaintext alphabet used in the substitution cipher
:type plaintext_alphabet: <class 'bytes'>
:param ciphertext_alphabet: Ciphertext alphabet used in the substitution cipher
:type ciphertext_alphabet: <class 'bytes'>
:return: Decoded string
:rtype: str
"""

decoded_string_list = []

for enc_str in bytes(encoded_string, "utf-8"):
if enc_str in ciphertext_alphabet:
decoded_string_list.append(chr(plaintext_alphabet[ciphertext_alphabet.find(enc_str)]))
else:
decoded_string_list.append(chr(enc_str))

return "".join(decoded_string_list)

```

```
def decode_config_strings(decrypted_config, plaintext_alphabet, ciphertext_alphabet,
config):
"""
This function implements the substitution cipher that Nighthawk uses.
Encoded strings are decoded.

:param decrypted_config: Decrypted Nighthawk config
:type decrypted_config: dict
:param plaintext_alphabet: Plaintext alphabet used in the substitution cipher
:type plaintext_alphabet: <class 'bytes'>
:param ciphertext_alphabet: Ciphertext alphabet used in the substitution cipher
:type ciphertext_alphabet: <class 'bytes'>
:return: JSON with decoded strings
:rtype: dict
"""

for k in decrypted_config:
decoded_string = _decode_str(k, plaintext_alphabet, ciphertext_alphabet)

if isinstance(decrypted_config[k], dict):
config[decoded_string] = decrypted_config[k].copy()
else:
config[decoded_string] = decrypted_config[k]
del config[k]

if isinstance(decrypted_config[k], dict):
config[decoded_string] = decode_config_strings(
decrypted_config[k], plaintext_alphabet, ciphertext_alphabet, config[decoded_string]
)
```

```
elif isinstance(decrypted_config[k], str):
config[decoded_string] = _decode_str(decrypted_config[k], plaintext_alphabet, ciphertext_alphabet)
if isinstance(decrypted_config[k], list):
config[decoded_string] = []
for s in decrypted_config[k]:
config[decoded_string].append(_decode_str(s, plaintext_alphabet, ciphertext_alphabet))

return config


def _get_section_data(data, section_name):
"""
Function to return data belonging to `section_name` section in PE `data`

:param data: Nighthawk DLL contents
:type data: <class 'bytes'>
:param section_name: Name of section whose data is to be retrieved
:type section_name: str
:return: section data
:rtype: <class 'bytes'> or None
"""

pe = None
with contextlib.suppress(Exception):
pe = pefile.PE(data=data, fast_load=False)

if pe is None:
return None

for section in pe.sections:
```

```
if section.Name.strip(b"\x00") == section_name:
return section.get_data()

return None


def _alphabet_heuristics(alphabets):
"""
This function implements heuristics to determine if an identified alphabet
string is actually an alphabet. These heuristics are purely based on my
observations.

:param alpha: Possible alphabet strings
:type alpha: list of <class 'bytes'>
:return: set of possible alphabet bytestrings
:rtype: set of <class 'bytes'>
"""

candidates = {}
finalists = set()

for alpha in alphabets:
num_whitespace = len(re.split(b"\s+", alpha))
if num_whitespace > 3:
# I've observed alphabets usually have num_whitespace == 2
continue

num_unique_chars = len(set(alpha))
if num_unique_chars < 15:
# I've observed that alphabets have large number of unique characters
# Random low threshold, though
continue

if num_unique_chars not in candidates:
candidates[num_unique_chars] = set()
candidates[num_unique_chars].add(alpha)

# I've observed that the plaintext and ciphertext alphabets both have the
# same number of num_unique_chars
for _, alphabets_ in candidates.items():
if len(alphabets_) > 1:
finalists.update(alphabets_)

return finalists


def get_possible_alphabet(data):
"""
Nighthawk is known to encode strings using a simple substitution cipher.
Decoding requires knowing the plaintext and ciphertext alphabets used.

:param data: Nighthawk DLL contents
:type data: <class 'bytes'>
:return: Permutation of possible plaintext and ciphertext alphabets
:rtype: <class 'itertools.permutations'> or None
"""

alphabets_regex = b"[\w\s!\\\"\#\$%\&'\(\)\*\+,\-\./:;<=>\?@\[\]\^_`\{\}\~\|]{86}\x00"
alphabets_regexc = re.compile(alphabets_regex)

# Alphabets are known to exist in the .rdata section, so just search there
```

```
rdata_data = _get_section_data(data, b".rdata")
matches = alphabets_regexc.findall(rdata_data)

if matches:
alphabets = _alphabet_heuristics(matches)
if alphabets:
# At this point, I have candidate alphabet strings but I don't know
# which is the plaintext alphabet and which is ciphertext alphabet
# To brute force, I'll calculate different permutations of length 2
return itertools.permutations(alphabets, 2)

return None


def decrypt_config(encrypted_config, decryption_key):
"""
Nighthawk config is gzip compressed and then encrypted with AES-128 CBC mode.

:param encrypted_config: Encrypted config data
:type encrypted_config: <class 'bytes'>
:param decryption_key: Config decryption key
:type decryption_key: <class 'bytes'>
:return: decrypted config
:rtype: dict or None
"""

cipher = AES.new(decryption_key, AES.MODE_CBC, IV=16 * b"\x00")
gzip_config = cipher.decrypt(encrypted_config)

if gzip_config[:2] != b"\x1F\x8B":
```

```
# gzip magic signature is b'\x1F\x8B' at offset 0
return None

# I've noticed gzip_config containing additional data at the end.
# Below statements truncate gzip_config to the rightmost b'\x00\x00'
# which is gzip end-of-stream marker
i = gzip_config.rindex(b"\x00\x00")
gzip_config = gzip_config[: i + 2]

config = gzip.decompress(gzip_config).decode("utf-8")
return json.loads(config)


def get_encoded_config(profile_section_contents):
"""
The contents of Nighthawk DLL .profile section contain 4 components:
1. Keying method
2. Config decryption key (optional)
2. Size of configuration
3. Encrypted configuration

At this point, it is confirmed that the keying method == 0 and config
decryption key is available in the .profile section.

:param data: Nighthawk DLL .profile section contents
:type data: <class 'bytes'>
:return: Encrypted config data
:rtype: <class 'bytes'> or None
"""

```

```
config_size = struct.unpack("<I", profile_section_contents[17:21])[0]
if config_size > (len(profile_section_contents) - 1 - 16 - 4):
# max config size == size of .profile section - keying method 1 byte - 16
# bytes config decryption key - 4 bytes config size field.
# Actual config size cannot be greater than max possible config size
return None

return profile_section_contents[21 : 21 + config_size]


def get_decryption_key(profile_section_contents):
"""
The contents of Nighthawk DLL .profile section contain 4 components:
1. Keying method
2. Config decryption key (optional)
2. Size of configuration
3. Encrypted configuration

:param data: Nighthawk DLL .profile section contents
:type data: <class 'bytes'>
:return: Config decryption key
:rtype: <class 'bytes'> or None
"""

keying_method = profile_section_contents[0]
if keying_method == 0:
# Config decryption key is embedded in .profile section contents
return profile_section_contents[1:17]

return None
```

```


def get_profile_section_contents(data):
"""
Nighthawk DLLs are known to contain a .profile section which contains
configuration information.

:param data: Nighthawk DLL contents
:type data: <class 'bytes'>
:return: .profile section contents
:rtype: <class 'bytes'> or None
"""

return _get_section_data(data, b".profile")


def extract_config(data):
"""
Configuration extractor for Nighthawk DLL

:param data: Nighthawk DLL contents
:type data: <class 'bytes'>
:return: Decrypted and decoded config
:rtype: dict or None
"""

# Will contain the final config that is passed to CAPEv2
cfg = {}

profile_section_contents = get_profile_section_contents(data)
if profile_section_contents is None:
```

```
return None

decryption_key = get_decryption_key(profile_section_contents)
if decryption_key is None:
return None

config = get_encoded_config(profile_section_contents)

decrypted_config = decrypt_config(config, decryption_key)

# decrypt_config is the decrypted configuration, but key and values strings
# are still encoded and need to be decoded. Nighthawk is known to encode
# strings using a simple substitution cipher. The real challenge is to extract
# the ciphertext and plaintext alphabet from the DLL

possible_alphabets = get_possible_alphabet(data)

for plaintext_alphabet, ciphertext_alphabet in possible_alphabets:
config_ = decode_config_strings(decrypted_config, plaintext_alphabet, ciphertext_alphabet, decrypted_config.copy())

if "implant-config" in config_:
# This is a heuristic and may fail in future versions
cfg["Plaintext Alphabet"] = plaintext_alphabet
cfg["Ciphertext Alphabet"] = ciphertext_alphabet
cfg["Config AES-128 CBC Decryption Key"] = decryption_key
cfg["Implant Config"] = config_
break

return cfg
```