latrodectus_static_unpacker.py

github.com/leandrofroes/malware-research/blob/main/Latrodectus/latrodectus_static_unpacker.py

leandrofroes

leandrofroes/**malware-research**

General malware analysis stuff

1 Contributor 0 Issues 35 Stars 4 Forks

1
2
3
4
5
6
7
8
9
10
11
12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

Author: Leandro Fróes

Date: 2024-05-03

Tested hashes

```
# aee22a35cbdac3f16c3ed742c0b1bfe9739a13469cf43b36fb2c63565111028c (Mar 4)

# 3b63ea8b6f9b2aa847faa11f6cd3eb281abd9b9cceedb570713c4d78a47de567 (Mar 11)

# 1625ac230aa5ca950573f3ba0b1a7bd4c7fbd3e3686f9ecd4a40f1504bf33a11 (Apr 25)

# 4cf2b612939359977df51a32d2f63e2cb0c6c601e114b8e4812bd548d1db85fe (Apr 26)

# 53e65d071870f127bc6bf6c8e8ddfd131558153513976744ee7460eeb766d081 (Apr 29)

# 1db686635bcdde30163e1e624c4d8f107fd2a20507690151c69cc6a0c482207a (Aug 15)

# NOTE: This is NOT a generic unpacker for Latrodectus family. This is

# just a tool I created for fun and that seems to work in some variants

# of the malware

# UPDATE (2024/05/09): It seems the packer used by those samples is a

# version of a known packer/crypter named 'Dave' that was reported by

# IBM some time ago.

# Reference: https://securityintelligence.com/x-force/trickbot-conti-crypters-where-are-they-now/

import re

import sys

import math

import pefile

import struct

from capstone import *

from collections import Counter

def xor_data(data: bytes, key: bytes) -> bytes:

"""

Perform a multibyte XOR operation against an array of bytes.

:param data: The bytes to be XORed.
```

```
:param key: The key to use in the XOR operation.

:return: The XORed bytes.

"""

result = bytearray()

for i in range(len(data)):

result.append(data[i] ^ key[i % len(key)])

return result

def search_key(target_addr: bytes, size: int) -> bytes:

"""

Try to find the XOR key used to decode the final payload.

:param target_addr: The address to look for the unpacking key.

:param size: The size of the block to be searched.

:return: The unpacking key found if any.

"""

cs = Cs(CS_ARCH_X86, CS_MODE_64)

cs.detail = True

cs.skipdata = True

key = b""

for inst in cs.disasm(target_addr, 0, size):

# If its not a MOV instruction anymore its probably the end of our key

if inst.mnemonic != "mov":

break

if inst.operands[0].type == 3 and inst.operands[1].access == 0:

value = b""

imm = inst.operands[1].value.imm
```

```
# Is there a better way to do this?
if inst.operands[1].size == 1:
value = struct.pack("<B", imm)
elif inst.operands[1].size == 2:
value = struct.pack("<H", imm)
elif inst.operands[1].size == 4:
value = struct.pack("<I", imm)
else:
break
key += value
# Usually the key has 16 bytes or less so we do this check to be safer
if len(key) >= 16:
break
# All the keys in the tested samples were null terminated so we make
# sure our key has it in case we missed it
if key[-1] != 0:
key += b"\x00"
if key:
print(f"[+] INFO: Found a potential key: '{key.replace(b"\x00", b"").decode()}'")
return key
def search_key_start_addr(data: bytes) -> list:
"""
Try to find the address containing the beginning of the key
construction on the stack.
:param data: The chunck of data to look for the start of the key
```

```
construction.

:return: A list containing all the potential start addresses for

the key construction.

"""

# The patterns we are looking for are one of the following:

# C6 44 24 ?? ??

# C7 44 24 ?? ?? ?? ?? ??

# We try to find some sequence of occurrences of those to be safer

rule = re.compile(b"(\xc6D\\$..){3,}|(\xc7D\\$.{5}){2,}")

matches = list(rule.finditer(data))

if not matches:

return []

target_chunks = []

for m in matches:

target_addr = m.start()

# There's no reason for using 256 here specifically, it's just

# a random number to provide some space for the chunck

target_chunk = data[target_addr:target_addr+256]

target_chunks.append(target_chunk)

return target_chunks

# Stolen from https://github.com/erocarrera/pefile/blob/master/pefile.py#L1324

def is_high_entropy(data: bytes) -> bool:

"""

Calculate the entropy of a chunk of data and determine if its high

or not based on a magic number (7 on this case)
```

```
:param data: The data chunk to calculate the entropy from.
:return: A boolean value indicating if the chunk entropy is high or not.
"""
occurences = Counter(bytearray(data))
entropy = 0
for x in occurences.values():
p_x = float(x) / len(data)
entropy -= p_x * math.log(p_x, 2)
if entropy > 6:
return True
else:
return False
def search_packed_resource(pe: pefile.PE) -> bytes:
"""
Try to find a potential resource containing the packed payload data.
:param pe: An instance of the pe file being analyzed.
:return: The packed resource found if any.
"""
if hasattr(pe, "DIRECTORY_ENTRY_RESOURCE"):
for entry in pe.DIRECTORY_ENTRY_RESOURCE.entries:
resource_type = pefile.RESOURCE_TYPE.get(entry.struct.Id)
# The tested samples that had the packed payload in a resource
# the resource type was an icon
if resource_type == "RT_ICON":
for directory in entry.directory.entries:
```

```
for resource in directory.directory.entries:

size = resource.data.struct.Size

# Some other simple checks to make sure its not

# just a regular icon

if size > 0x9000:

offset = resource.data.struct.OffsetToData

data = pe.get_data(offset, size)

if is_high_entropy(data):

print("[+] INFO: Found a pontential packed payload in the resources.")

return data

return None

def main():

if len(sys.argv) != 2:

print(f"Usage: {sys.argv[0]} <file>")

sys.exit(1)

filename = sys.argv[1]

print(f"[+] INFO: Starting analysis for '{filename}'.")

pe = pefile.PE(filename)

packed_payload = b""

packed_section_name = ""

text_sec_data = None

text_sec_rva = 0

# We maintain a list of common section names to exclude in our

# search for 'weird section names'

common_section_names = [b".text", b".data", b".rdata", b".pdata", b".reloc", b".rsrc"]
```

```
for section in pe.sections:
# Get .text section info for further usage
if section.Name[:5] == b".text":
text_sec_data = section.get_data()
text_sec_rva = section.VirtualAddress
# Check if the packed payload is in a section with a 'weird name'
if section.Name not in common_section_names:
packed_payload = section.get_data()
packed_section_name = section.Name.replace(b"\x00", b"").decode()
assert text_sec_data is not None
assert text_sec_rva != 0
# If the payload was not found in the sections we try to find it in
# the resources
if not packed_payload:
packed_payload = search_packed_resource(pe)
else:
print(f"[+] INFO: Found a potential packed payload in the '{packed_section_name}' section.")
# If we reach this point then we found nothing at all and exit
if not packed_payload:
print("[!] ERROR: Unable to find the packed payload.")
sys.exit(1)
# The key is built in the stack and is very close to the entrypoint
# so we start from there
offset = pe.OPTIONAL_HEADER.AddressOfEntryPoint - text_sec_rva
# Since the string is built pretty close to the entrypoint we'll
```

```
# analyze a small chunk starting from there as an attempt to be
# more precise. Yes, it's easy to bypass, but works for now :)
size = 0x400
start_chunk = text_sec_data[offset:offset+size]
# Try to find the start address to look for the key
target_chunks = search_key_start_addr(start_chunk)
if not target_chunks:
print("[!] ERROR: Unable to find the key chunk in the stack.")
sys.exit(1)
final_payload = b""
unpacked = False
for chunk in target_chunks:
key = search_key(chunk, len(chunk))
if not key:
print("[!] ERROR: Unable to find the key.")
sys.exit(1)
# Try to 'decode' the payload using the key found
final_payload = xor_data(packed_payload, key)
# Loosy check in the DOS signature to make sure we have a 'valid PE'
if final_payload[0] == 77 and final_payload[1] == 90:
out_file = filename + "_unpacked"
with open(out_file, "wb+") as f:
f.write(final_payload)
unpacked = True
print(f"[+] INFO: Payload unpacked successfully and written to {out_file}")
```

```
break

else:

print("[!] INFO: Unable to decode the payload with the key found.")

if not unpacked:

print("[!] ERROR: None of the keys found were able to unpack the payload.")

sys.exit(1)

print("[+] Done!")

if __name__ == "__main__":

main()
```