Decrypting and Hunting PrivateLoader

By André Tavares

Published: 2022-06-06 · Archived: 2026-04-10 02:07:58 UTC

PrivateLoader is a loader from a pay-per-install malware distribution service that has been utilized to distribute info stealers, banking trojans, loaders, spambots, rats, miners and ransomware on Windows machines. First seen in early 2021, being hosted on websites that claim to provide cracked software, the customers of the service are able to selectively deliver malware to victims based on location, financial activity, environment, and specific software installed.

Let's have a look at the malware and try to find a way to detect and hunt it.

Searching for strings

Here's a sample analyzed by Zscaler on April 2022:

`aa2c0a9e34f9fa4cbf1780d757cc84f32a8bd005142012e91a6888167f80f4d5`

Let's open it on Ghidra. Going into the entry point, following the code, looking for interesting functions, I quickly spot the function at `0x406360`. It's calling `LoadLibraryA` but the `lpLibFileName` parameter is built dynamically at runtime using the stack. Its seems that we found a string encryption technique. Both the string and the xor key are loaded into the stack. Looking a bit more through the function, its seems that this is the way most of the strings are loaded:

```
 LEA       EAX=>local_50,[ESP + 0x10]
MOV       dword ptr [ESP + local_50[0]],0x84038676
MOV       dword ptr [ESP + local_50[4]],0xeb71eb3c
MOV       dword ptr [ESP + local_50[8]],0x36fb7b30
MOV       dword ptr [ESP + local_50[12]],0xab7d1f0c
MOVAPS    XMM1,xmmword ptr [ESP + local_50[0]]
MOV       dword ptr [ESP + local_30[0]],0xea71e31d
MOV       dword ptr [ESP + local_30[4]],0xd9428759
MOV       dword ptr [ESP + local_30[8]],0x5a971f1e
MOV       dword ptr [ESP + local_30[12]],0xab7d1f0c
PXOR      XMM1,xmmword ptr [ESP + local_30[0]] ; kernel32.dll
PUSH      EAX  ; LPCSTR lpLibFileName for LoadLibraryA
MOVAPS    xmmword ptr [ESP + local_50[0]],XMM1
CALL      ESI=>KERNEL32.DLL::LoadLibraryA
```

After XOR the encrypted string with the key, we get `kernel32.dll`.

Decrypting the strings

Now, to faster analyze the malware and better understand its behavior, we should build a string decryptor to help us on our reversing efforts and better document the code. With the help of Capstone disassembly framework, and some trial and error, here's the script:

```
import pefile
import struct
from capstone import *

def extract_var(op):
if ']' in op:
op = ''.join(op.split(' ')[-1])[:-1].replace('[', '')
return op

def search(instructions, var):
data_chunks = []
for inst in instructions:
if inst[2] == 'mov':
try:
imm = int(inst[3].split(' ')[-1], 16)
data_chunks.append(struct.pack('<I', imm))
if extract_var(inst[3].split(', ')[0]) == var:
return b''.join(data_chunks[::-1]) # 16 bytes str chunk
except: # not a dword
pass

if extract_var(inst[3].split(', ')[0]) == var:
var = extract_var(inst[3].split(', ')[1])

def decrypt_strings(filename):
# disassemble .text section
pe = pefile.PE(filename)
```

```
md = Cs(CS_ARCH_X86, CS_MODE_32)
md.skipdata = True
instructions = []
text = pe.sections[0]
text_addr = pe.OPTIONAL_HEADER.ImageBase + text.VirtualAddress
for (addr, size, mnemonic, op_str) in md.disasm_lite(text.get_data(), text_addr):
instructions.append((addr, size, mnemonic, op_str))

# search, build and decrypt strings
strings = []
addr = None
string = ''
for i, inst in enumerate(instructions):
if inst[2] == 'pxor':
try: # possible string decryption found
encrypted_str = search(instructions[:i][::-1], extract_var(inst[3].split(', ')[0])) # reverse search
key = search(instructions[:i][::-1], extract_var(inst[3].split(', ')[1])) # reverse search
string += bytearray(encrypted_str[j] ^ key[j] for j in range(len(key))).decode(errors='ignore') # bug

if not addr:
addr = hex(inst[0])

if '\x00' in string:
strings.append((addr, string.replace('\x00', '')))
string = ''
addr = None

except Exception as e:
print(f'Fail at {hex(inst[0])}')
```

```
    print(len(strings))
    for s in strings:
        print(f'{s[0]} {s[1]}')

    decrypt_strings('aa2c0a9e34f9fa4cbf1780d757cc84f32a8bd005142012e91a6888167f80f4d5')
```

After running it against the sample we are analyzing, we get the following strings:

```
 0x4003ee GetCurrentProcess
0x400469 CreateThread
0x4004ba CreateFileA
0x400506 Sleep
0x400572 SetPriorityClass
0x4005ec Shell32.dll
0x400657 SHGetFolderPathA
0x40083b null
0x401078 rb
0x40157c http://212.193.30.45/proxies.txt
0x401795 :1080
0x401839 \n
0x401f2d :1080
0x401fd1 :
0x4026ce .
0x4028ac .
0x402972 .
0x402a34 .
0x4032ad http://45.144.225.57/server.txt
0x4033c0 HOST:
0x40346e :
0x403760 pastebin.com/raw/A7dSG1te
0x403965 HOST:
0x403b93 http://wfsdragon.ru/api/setStats.php
0x403dcd HOST:
0x403f84 :
0x4040ae 2.56.59.42
0x404350 /base/api/statistics.php
0x404439 URL:
0x4044b6 :
0x404a5e https://
0x404ad8 .tmp
0x404bf6 \
0x4053e9 kernel32.dll
0x40544a WINHTTP.dll
0x4054a5 wininet.dll
```

```
0x406616 WinHttpConnect
0x406682 WinHttpOpenRequest
0x40671a WinHttpQueryDataAvailable
0x4067b2 WinHttpSendRequest
0x40684a WinHttpReceiveResponse
0x4068e2 WinHttpQueryHeaders
0x406956 WinHttpOpen
0x4069b5 WinHttpReadData
0x406a20 WinHttpCloseHandle
0x406b09 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.63 Sa
0x407402 http://
0x4074ab /
0x407582 ?
0x40851a HEAD
0x408fa8 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.63 Sa
0x4091f0 wininet.dll
0x40925b InternetSetOptionA
0x4092ef HttpOpenRequestA
0x40938d InternetConnectA
0x409421 InternetOpenUrlA
0x40949e InternetOpenA
0x4094f2 HttpQueryInfoA
0x409567 InternetQueryOptionA
0x4095fb HttpSendRequestA
0x409694 InternetReadFile
0x409737 InternetCloseHandle
0x4097ad Kernel32.dll
0x409801 HeapAlloc
0x409852 HeapFree
0x4098a3 GetProcessHeap
0x4098f3 CharNextA
0x409938 User32.dll
0x409994 GetLastError
0x4099e5 CreateFileA
0x409a36 WriteFile
0x409a87 CloseHandle
```

Some of them are network IoCs that can be used for defense and tracking purposes. We can now go back to Ghidra and continue our analysis, now with more context of what might be the malware's capabilities.

Detecting and hunting the malware

This uncommon string decryption technique enable us to write a Yara rule for detection and hunting purposes. To reduce the number of false positives and increase the rule performance, we can add some plaintext unicode strings used on the C2 communication and a few minor conditions. Here's the rule:

```
rule win_privateloader : loader
{
meta:
author = "andretavare5"
org = "BitSight"
date = "2022-06-06"
md5 = "8f70a0f45532261cb4df2800b141551d"
reference = "https://tavares.re/blog/2022/06/06/hunting-privateloader-pay-per-install-service"
license = "CC BY-NC-SA 4.0"

strings:
$x = {66 0F EF (4?|8?)} // pxor xmm(1/0) - str chunk decryption
$s = "Content-Type: application/x-www-form-urlencoded\r\n" wide ascii
$ua1 = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169 Safari/537.36" wide ascii
$ua2 = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.63 Safari/537.36" wide ascii

condition:
uint16(0) == 0x5A4D and // MZ
$s and
any of ($ua*) and
#x > 100
}
```

After running this rule on VirusTotal retro hunting, I got over 1k samples on a 1 year timeframe. By manually analyzing some of the matches, I couldn’t find any false positives. As a first attempt of hunting and detecting PrivateLoader, this rule seems to yield good results.
