LusyPOS and Tor

Published: 2014-12-01 · Archived: 2026-04-02 11:01:40 UTC

Introduction

At our dayjobs, as reverse engineers at CBTS, Jeremy and I have been hunting new POS malware.

A new sample appeared on Virustotal this week that had a very interesting name “lusypos.exe”. There have been very few references to this particular family and it appears to be fairly new. Google searching was able to give me the following information:

CRDCLUB.WS / Кардинг Форум / Carding Forum
www.crdclub.ws/ ▾ Translate this page
LusyPOS malware(dumps grabber) - Pro's... 134, 7, HouseOfCarders ▻, 28-11, 06:23,
UNVERIFIED ADVERTISEMENT. Old, Приму звонки/смс от контор, 3, 1 ...

The sample that I’ll be talking about in this post is bc7bf2584e3b039155265642268c94c7.

At the time of this writing the malware is currently flagged on Virustotal by 7/54 engines.

SHA256:	d7a08338bcb30cc688a827b611fe9b26c54f3ba35c02355fa1d468da8cbbd903
File name:	lusypos.exe
Detection ratio:	7 / 54
Analysis date:	2014-11-30 02:59:29 UTC (1 day, 12 hours ago)

Interestingly, some of the signatures seem to be hitting on the copy of tor.exe that is in the bundle.

Antiy-AVL	RiskWare[NetTool:not-a-virus]/Win32.Tor	20141129
Avira	TR/Hijacker.Gen	20141129
DrWeb	Trojan.Packed.21724	20141130
Kaspersky	not-a-virus:NetTool.Win32.Tor.v	20141130
NANO-Antivirus	Trojan.Win32.Chgt.deuqtw	20141130
Panda	Trj/Genetic.gen	20141129
VBA32	Malware-Cryptor.Inject.gen	20141128

Analysis

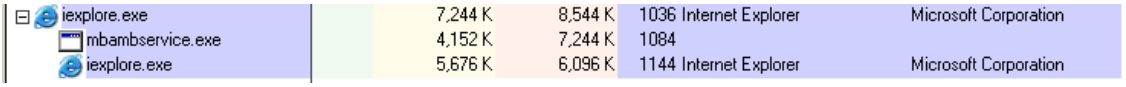
This malware clocks in around 4.0 MB in size, so it’s not small. For comparison, getmypass POS malware was 17k in size.

The first thing of note when executing this in a sandbox is that this malware drops a copy of tor.exe, libcurl.dll, and zlib1.dll. It also copies itself to the %APPDATA% directory on the victim host. The following are the locations and MD5's of the dropped files are below:

The file mbambservice.exe is the copy of tor.exe

```
 d0f3b3aaa109a1ea8978c83d23055eb1  C:\Documents and Settings\<USER>\Application Data\VeriFone32\libcurl.dll
4407393c1542782bac2ba9d017f27dc9  C:\Documents and Settings\<USER>\Application Data\VeriFone32\mbambservice.exe
bc7bf2584e3b039155265642268c94c7  C:\Documents and Settings\<USER>\Application Data\VeriFone32\verifone32.exe
b8a9e91134e7c89440a0f95470d5e47b  C:\Documents and Settings\<USER>\Application Data\VeriFone32\zlib1.dll
```

The malware will also create the mutex "prowin32Mutex" and injects code into iexplore.exe. This was a strange mix of dexter-like behavior mixed with Chewbacca-like techniques.



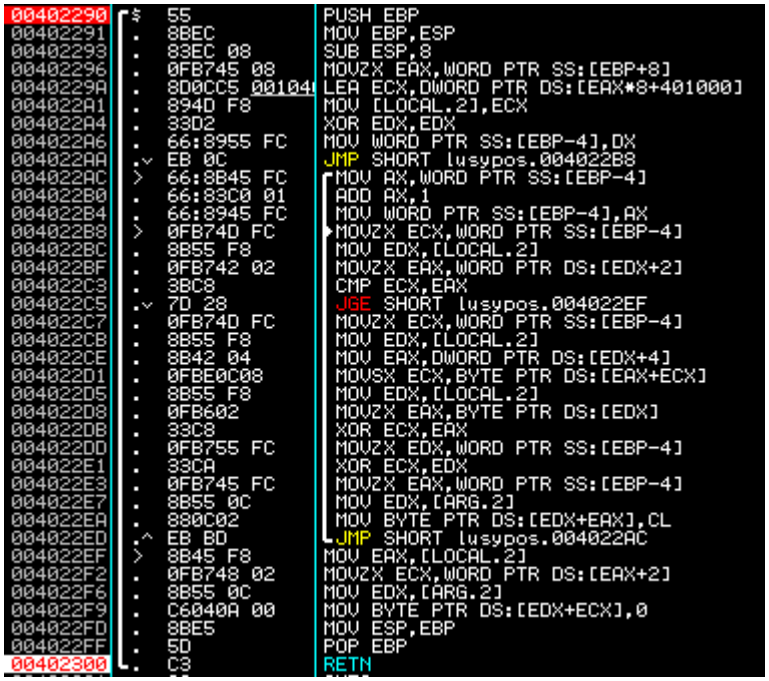
While running in a sandbox, the malware communicated out to

```
 86.59.21.38
212.112.245.170
128.31.0.39
154.35.32.5
193.23.244.244
```

Now let's get to the good stuff.

Decoding

The malware has an interesting method of decoding strings that are statically defined in the binary.



For the non-asm folks on here, the malware is using a lookup table with structures containing a one byte xor key, pointer to the string, and length of the string. It will perform an additional xor operation at the end.

A decoder for this is written (in python below)

```
 #!/usr/bin/env python

# ------------------------------------------------------------------------------ #
# Author:       Jeremy Humble - CBTS ACS
# Description:  POC LusyPOC String Extractor. Strings are stored in an array
#               of 8 byte structs with the following structure: {short xor_key,
#               short length, char* encoded_string}
# ------------------------------------------------------------------------------ #

import sys
import struct
import binascii
import pefile
import simplejson as json

from pprint import pprint
from optparse import OptionParser

# Option Parsing
usage = "lusypos_parser.py [-j] lusypos_sample1 [lusypos_sample2] ..."
opt_parser = OptionParser(usage=usage)
opt_parser.add_option("-j", "--json", action="store_true",dest="json_output",
   help="Output all information on each string in json format")
opt_parser.add_option("-p", "--pretty", action="store_true",dest="pretty_json_output",
```

```
   help="Output all information on each string in pretty json format")
(options, args) = opt_parser.parse_args()

if options.json_output and options.pretty_json_output:
    sys.stderr.write('Use either -j or -p, not both')
    exit()

class LusyEncodedString:

    def __init__(self,raw_data,file_content,pe):
        self.xor_key = struct.unpack('H',raw_data[0:2])[0]
        self.length = struct.unpack('H',raw_data[2:4])[0]
        self.virtual_offset = struct.unpack('I', raw_data[4:8])[0]
        self.raw_offset = pe.get_offset_from_rva(self.virtual_offset - pe.OPTIONAL_HEADER.ImageBas
        self.encoded_str = file_content[self.raw_offset:self.raw_offset+self.length]
        self._decode()

    def _decode(self):
        self.decoded_str = ""
        for i in range(0,self.length):
            self.decoded_str += chr(ord(self.encoded_str[i]) ^ self.xor_key ^ i)

    def __str__(self):
        return str(self.to_dict())

    def to_dict(self):
        d = {'xor key': hex(self.xor_key), 'length': self.length, 'raw offset': hex(self.raw_offs
             'virtual offset': hex(self.virtual_offset), 'encoded string': self.encoded_str, 'dec
        return d



# For now we'll assume the table is always at RVA 401000 (raw 0x400) as hardcoded in bc7bf2584e3b0
# With a little more refinement this could probably be found dynamically. AFAIK it's always locate
# Until I see a sample that shows otherwise, there's no point in doing this
def parse_table(content,pe,table_rva=0x1000):
    encoded_strings = []
    raw_offset = pe.get_physical_by_rva(table_rva)
    i = 0
    while True:
        raw_struct = content[raw_offset+i*8:raw_offset+i*8+8]
        # The last struct in the table is all null bytes. Stop parsing when we hit it
        if struct.unpack('<Q',raw_struct)[0] == 0:
            break
        else:
            try:
                encoded_strings.append(LusyEncodedString(raw_struct,content,pe))
```

```
            except Exception as e:
                sys.stderr.write('Error processing entry "%s" with Exception "%s". Ending'
                                 ' table processing\n' % (binascii.hexlify(raw_struct),e))
        i += 1
    return encoded_strings

if __name__ == '__main__':
    fname_to_lusy_string_map = {}
    for arg in args:
        try:
            pe = pefile.PE(arg)
            with open(arg,'r') as fp:
                content = fp.read()
                fname_to_lusy_string_map[arg] = parse_table(content,pe)
        except Exception as e:
            sys.stderr.write('Exception processing file %s: "%s"\n' % (arg,e))

    if options.json_output or options.pretty_json_output:
        json_dict = {}
        # Call to_dict on all of the objects so we can dump json
        for fname, lusy_strings in fname_to_lusy_string_map.items():
            json_dict[fname] = []
            for lusy_str in lusy_strings:
                json_dict[fname].append(lusy_str.to_dict())
        # If only working on one file, omit the top level filename key since it's obvious
        if len(json_dict.keys()) == 1:
            json_dict = json_dict[json_dict.keys()[0]]
        if options.json_output:
            print json.dumps(json_dict)
        else:
            pprint(json_dict)
    else:
        for fname, lusy_strings in fname_to_lusy_string_map.items():
            for lusy_str in lusy_strings:
                print lusy_str.decoded_str
```

Which when executed will decode the following strings:

```
 http://kcdjqxk4jjwzjopq.onion/d/gw.php
http://ydoapqgxeqmvsugz.onion/d/gw.php
VeriFone32
verifone32
prowin32Mutex
b00n v1.1
\\Internet Explorer\\iexplore.exe
mbambservice.exe
```

```
tor.exe
zlib1.dll
libcurl.dll
Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\Associations
Software\\Microsoft\\Windows\\CurrentVersion\\Internet Settings\\Zones\\0
LowRiskFileTypes
Content-Type: application/x-www-form-urlencoded
127.0.0.1:9050
Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64; Trident/6.0) g00n
curl_easy_init
curl_easy_setopt
curl_easy_cleanup
curl_easy_perform
curl_easy_strerror
curl_slist_append
curl_easy_getinfo
curl_slist_free_all
page=
&ump=
&ks=
&opt=
&unm=
&cnm=
&view=
&spec=
&query=
&val=
&var=
DetectShutdownClass
download-
update-
checkin:
scanin:
uninstall
response=
UpdateMutex:
Software\\Verifone32
Software\\Microsoft\\Windows\\CurrentVersion\\Run
.DEFAULT\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
mbambservice.exe
wmiprvse.exe
LogonUI.exe
svchost.exe
iexplore.exe
explorer.exe
System
smss.exe
```

```
csrss.exe
winlogon.exe
lsass.exe
spoolsv.exe
alg.exe
wuauclt.exe
firefox.exe
chrome.exe
devenv.exe
```

This contains the C2 information, along with a process whitelist, and registry keys for persistence. One thing to note based on these strings, is that it looks like the malware may have taken a cue from dexter.

RAM Scraping

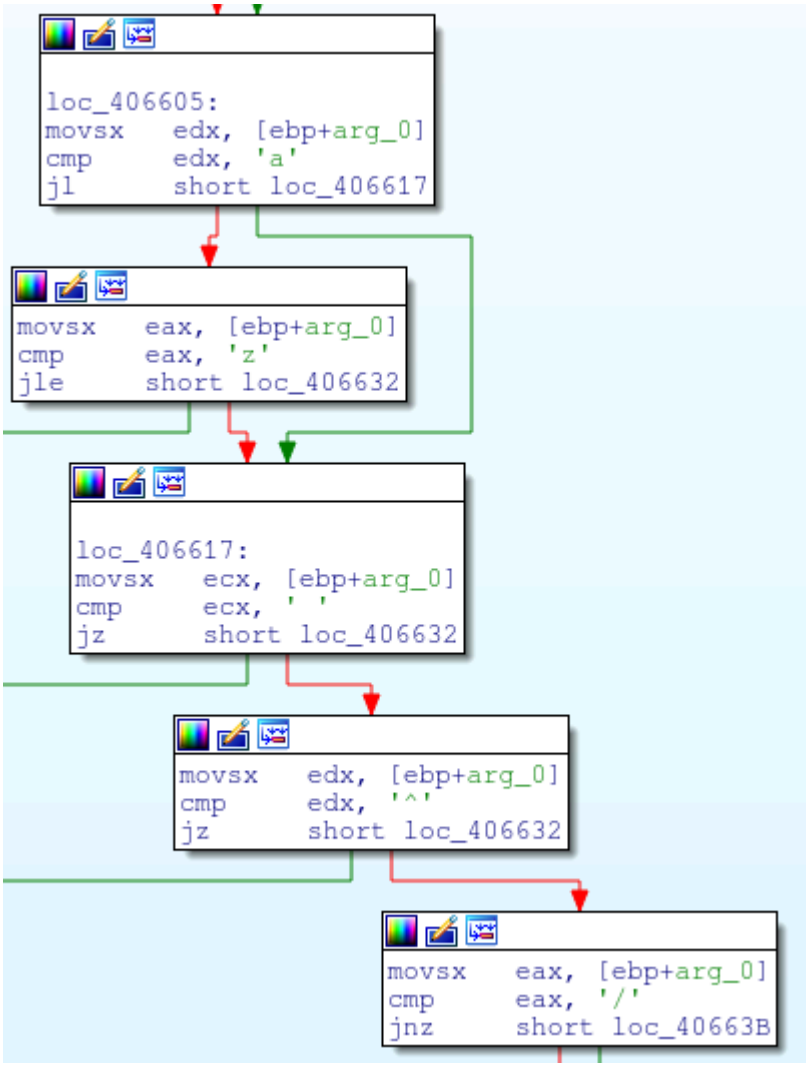
RAM scraping is performed through the common sequence of using CreateToolhelp32Snapshot, then using Process32First and Process32Next to iterate. Pseudocode for that would look something like the following:

```
 handle = CreateToolhelp32Snapshot
Process32First(handle)
do
         sleep 1000
         OpenProcess
         VirtualQueryEx
         ReadProcessMemory
         CloseHandle
         Sleep 5000
while Process32Next
```

This technique is not new and is commonly used in many different POS Ram scrapers. Truth is, that without writing a driver, the malware authors often have their hands tied and only have a few techniques to peer into process memory space.

CC Validation

The malware also contains methods to search memory for sequences of data that look like credit card track information.



Once it finds that data, there are checks against the potential credit card number to determine if it is Luhn valid. Luhn's algorithm is the defacto algorithm for validating credit card numbers. It can be seen implemented in the malware using a lookup table rather than calcuating the digital root. One note, is that this is the same implementation of Luhn's as FrameworkPOS, Dexter, and getmypass.

```
v3 = 0;
v4 = 2;
v5 = 4;
v6 = 6;
v7 = 8;
v8 = 1;
v9 = 3;
v10 = 5;
v11 = 7;
v12 = 9;
v18 = 1;
v17 = 0;
v19 = a2;
while ( 1 )
{
  v13 = v19--;
  if ( !v13 )
    break;
  v15 = *(_BYTE *)(v19 + a1) - 48;
  if ( v18 )
    v14 = v15;
  else
    v14 = *(&v3 + v15);
  v17 += v14;
  v16 = v18 == 0;
  v18 = v18 == 0;
}
return v17 % 10 == 0;
```

Closing Thoughts

When looking into malware families like Chewbacca and now LusyPOS, one thought comes to mind. Why would a POS machine be allowed to talk to tor? Most PCI audits will attempt to lock this sort of activity down, but there seems to be devils in the implementation that allow malware like this to be successful.

This is just a scratch in the surface of a new malware family. We'll be curious to watch it evolve over the next couple years and track its progress.

Source: https://securitykitten.github.io/2014/12/01/lusypos-and-tor.html