Diving into Pluroxs DNS based protection layer

sysopfb.github.io/malware,/crypters/2019/09/23/Plurox-packer-layer-unpacked.html

Random RE September 23, 2019

Sep 23, 2019

Intro

Recently saw someone mentioning a sample of Plurox performing code flow obfuscation based on the result of a DNS request, kind of interesting and I have apparently lost the link to the person that originally mentioned the hash… so if you recognize it let me know and I'll update this post.

The file we'll be looking at is
0385038427750543d98ce02a2a24aef45a937ef226a53fc3f995b5cea513b1c8

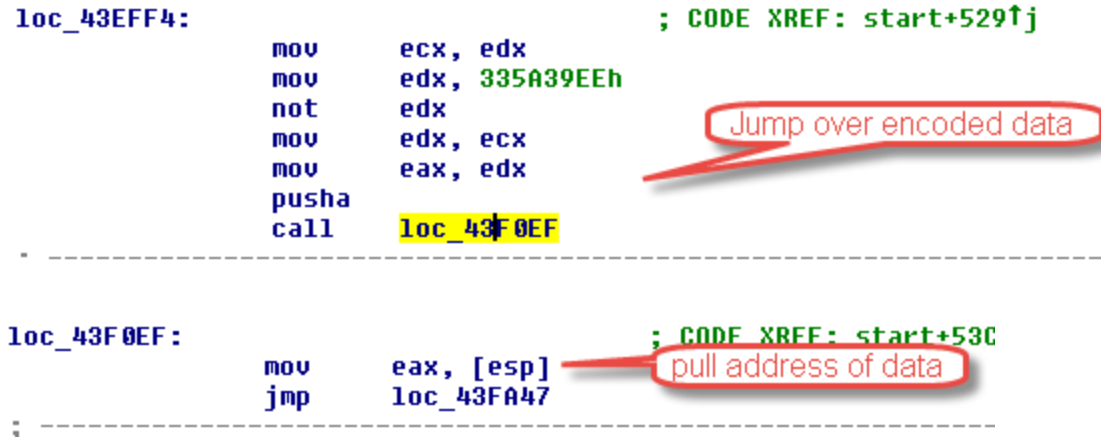
PDB:

```
E:\OldSoftware\Generating\Crypto\crypto.pdb
```

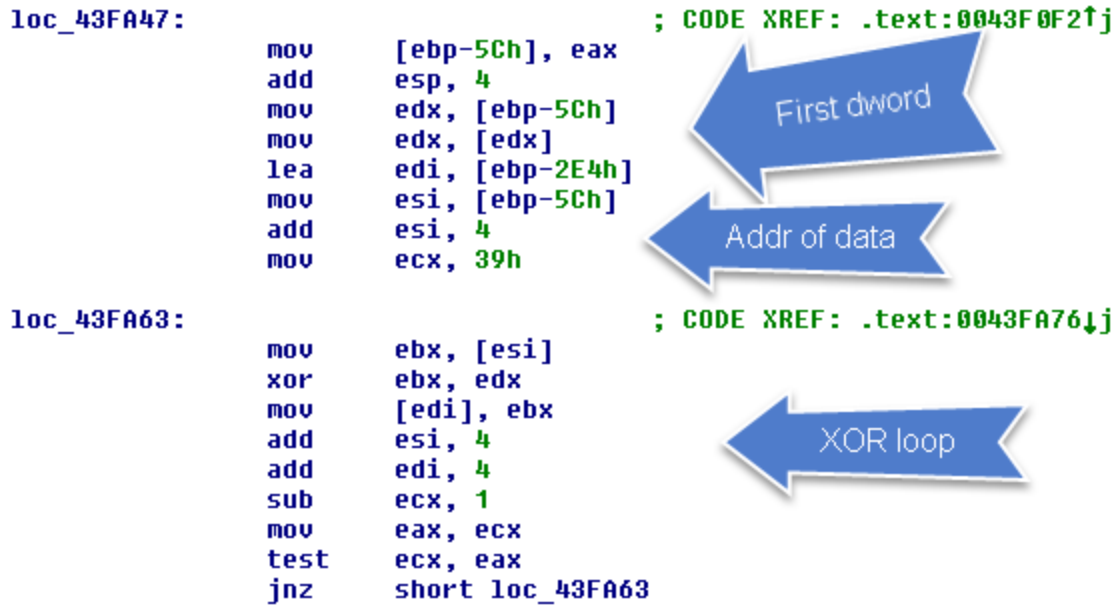
Strings

Encoded strings is pretty common in malware and especially in protection layers such as crypters/packers and droppers/loaders.

Near the entry point of the sample we'll find a call instruction with an immediate access of the stack. You find this pretty frequently in frameworks that use shellcode such as Metasploit, it's basically getting the address of the data immediately following the call instruction.



What are we doing with this data? Further down we find a loop using XOR while loading the first DWORD of the data as the XOR key.

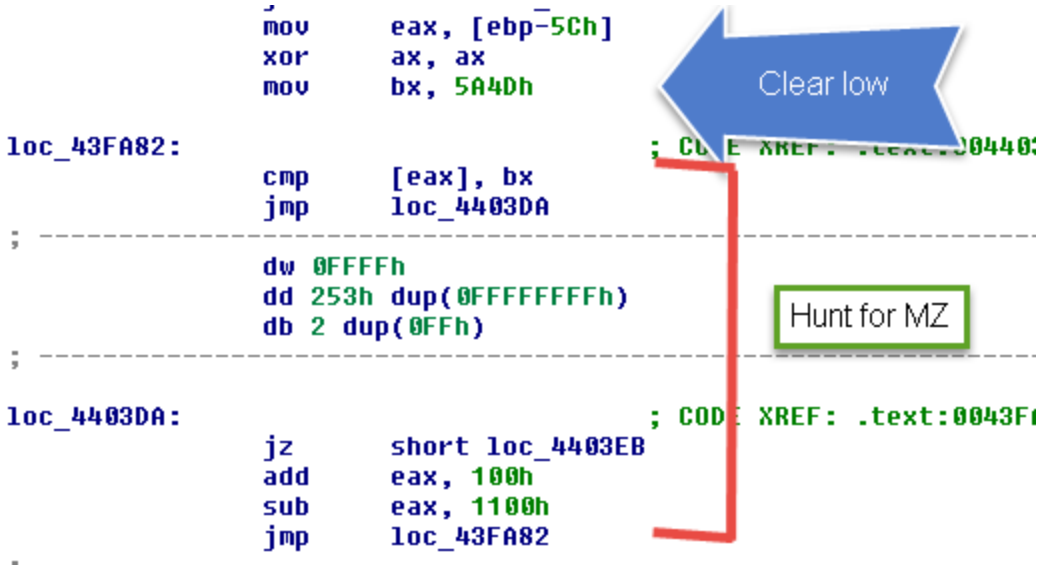


Doing this statically in python:

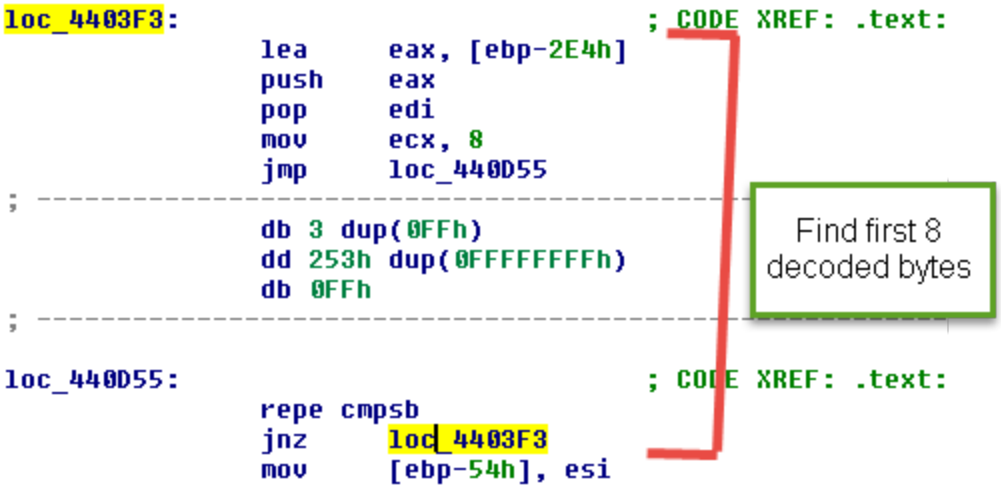
```
Python>data = GetManyBytes(0x43f007, 0x100)
Python>key = data[:4]
Python>data = data[4:]
Python>key = bytearray(key)
Python>data = bytearray(data)
Python>for i in range(len(data)):
Python>  data[i] ^= key[i%len(key)]
Python>
Python>data
AZ$9R9GetProcAddress
Python>data.split('\x00')
[bytearray(b'\x12\xf1\x0e\x03AZ\xf1$9\x99\xaf\xfaR\x039\x98GetProcAddress'),
bytearray(b'VirtualFree'), bytearray(b'UnmapViewOfFile'), bytearray(b'htonl'),
bytearray(b'ntdll.dll'), bytearray(b'ws2_32.dll'), bytearray(b'ExitProcess'),
bytearray(b'gethostbyname'), bytearray(b'LoadLibraryA'), bytearray(b'VirtualAlloc'),
bytearray(b'User32.dll'), bytearray(b'RtlDecompressBuffer'),
bytearray(b'WSAStartup'), bytearray(b'VirtualProtect'), bytearray(b'google-public-
dns-b.google.com'), bytearray(b''), bytearray(b''), bytearray(b'\x8f'),
bytearray(b',\xe1T\r\x08\x08\xfb\xfb\xf7\xf7\xfb\xfb\xf7\xf7\xfb\xfb\xf7\xf7\xfb\xfb\x
```

This is our string block along with the domain that will be resolved, the function names and domain immediately make me think this is more related to a protection or crypter layer as opposed to directly associated with the underlying malware Plurox. Assumptions however are just things that need to be proven.

Continuing on with the code it will enumerate it's own memory space looking for the start value 'MZ'.



After finding that address it begins utilizing the bytes at the beginning of the decoded string block, turns out these values are headers used to identify two blocks of data that will be saved off.



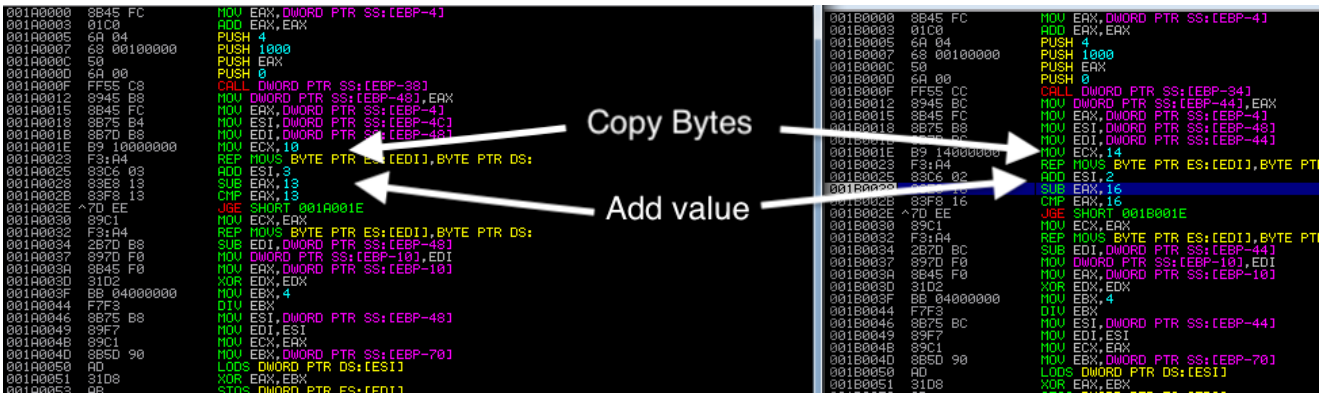
After saving off those chunks of data it will perform the resolution of the domain from the decoded strings.

```
mov     edi, [ebp-58h]
rep movsb               ; First block found copied into second block allocated
mov     ecx, [ebp-4]
mov     esi, [ebp-50h]
mov     edi, [ebp-4Ch]
rep movsb               ; Second larger block found copied into first block allocated
lea     eax, [ebp-200h]
push    eax
push    202h
call    dword ptr [ebp-1Ch] ; WSAStartup
lea     eax, [ebp-221h] ; google-public-dns-b.google.com
push    eax
call    dword ptr [ebp-18h] ; gethostbyname
test    eax, eax
jnz     short loc_442B5C
push    0
call    dword ptr [ebp-28h] ; ExitProcess

                        ; CODE XREF: .text:00442B55↑j
mov     eax, [eax+0Ch]
mov     eax, [eax]
mov     eax, [eax]
push    eax
call    dword ptr [ebp-14h] ; htonl
mov     ecx, eax
mov     [ebp-70h], ecx  ; Store resolved IP - 0x08080404
mov     esi, [ebp-58h]
jmp     loc_4434C4
```

It will then use that resolved IP as the DWORD XOR key for the smaller layer that was previously saved off.

```
loc_443E1C:                             ; CODE XREF: .text:004434C7↑j
                mov     ecx, [ebp-8]
                shr     ecx, 2

loc_443E22:                             ; CODE XREF: .text:00443E31↓j
                mov     eax, [esi]
                add     esi, 4
                mov     ebx, [ebp-70h]
                xor     eax, ebx        ; XORs by resolved IP
                mov     [edi], eax
                add     edi, 4
                loop    loc_443E22
                jmp     dword ptr [ebp-58h]
```

This layer is designed to rebuild the larger blob that was previously saved off and then XOR decoded and LZNT decompressed if needed. Unfortunately the values used to rebuild are not static, you will need the number of bytes to copy over and the number of bytes to skip. Pulling out these values is possible through a number of ways but probably the easiest is to simply regex them out of the decoded layer.



So for creating an unpacker we need to do the following steps:

1. Load the PE file in memory mapped form
2. Get the OEP (entry point)
3. Find the encoded blob of strings
4. Find the two data blobs using the byte chunks from the decoded strings
5. Find the XOR key
6. Decode the second layer code
7. Get the value for bytes copied and bytes skipped
8. Rebuild the encoded payload using values from 7
9. Decode rebuilt payload
10. Check if compressed
11. Write to disk

Some of the pieces from above will be quickly glossed over because they are self explanatory, if these pieces are more technically advanced than you are ready for you can skip them by all means and just read the comments that way you can hopefully understand my thought process while I constructed the code which could be beneficial while learning.

For loading the PE file into memory and getting the OEP we will use pefile in python.

```
if __name__ == "__main__":
        fdata = open(sys.argv[1], 'rb').read()
        pe = pefile.PE(data=fdata)
        oep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
        found = False
        memmapped = pe.get_memory_mapped_image()
```

At this point we want to try to limit our scope, the encoded blob of strings appears to normally be near the OEP so we can probably try to brute it out within a limited scope of bytes.

```
        data = memmapped[oep:oep+0x1000]
        for i in range(len(data)-24):
                test_k = bytearray(data[i:i+4])
                test_v = bytearray(data[i+4:i+0x100])
                for j in range(len(test_v)):
                        test_v[j] ^= test_k[j%len(test_k)]
                if 'Alloc' in test_v or 'Process' in test_v or 'Decompress' in
test_v:
                        print("Found it")
                        found = True
                        break
```

So we are looping through the first 0x1000 bytes after the OEP to try to find encoded string bytes. After finding the start we just need to XOR decode the chunk which is semi redundant because we just did it for our testing loop and then split up the strings. Technically we only need the first 16 bytes but I figured it'd be good to add in code that can quickly parse and dump the strings to look for the different domains being used.

```
        if found == True:
                blob = bytearray(data[i+4:i+0x100])
                key = bytearray(data[i:i+4])
                for i in range(len(blob)):
                        blob[i] ^= key[i%len(key)]
                if '\x00\x00\x00' in blob:
                        conf = str(blob.split('\x00\x00\x00')[0]).split('\x00')
                else:
                        conf = str(blob).split('\x00')[:-1]
```

Next we will pull out the two blobs using the first 16 bytes from the decoded strings

```
                off1 = fdata.find(conf[0][:8])
                l = fdata[off1+8:].find(conf[0][:8])
                blob1 = fdata[off1+8:off1+l+8]

                off2 = fdata.find(conf[0][8:16])
                l = fdata[off2+8:].find(conf[0][8:16])
                blob2 = fdata[off2+8:off2+l+8]
```

Now we're going to do something a little interesting, this is why I enjoy writing scripts like this also. After gathering enough samples and dumping enough layer2s we can start to see byte patterns emerge, now we could just resolve the domain and boom we have our XOR key but that's no fun. So if you go back up to the comparison picture you'll notice some overlap of bytes in the decoded layer2 but more importantly I noticed the 4 bytes after the first 3 bytes remain pretty static across many samples tested.

```
                known_val = bytearray('\x01\xc0\x6a\x04')
```

So to get the XOR key we just need to XOR the known value with the encoded bytes in the same place and then fixup the key position because we went 3 bytes in instead of 4.

```
                key = bytearray(blob1[3:7])
                for i in range(len(key)):
                        key[i] ^= known_val[i]
                key = key[1:]+key[:1]
                temp = bytearray(blob1)
                for i in range(len(temp)):
                        temp[i] ^= key[i%len(key)]
```

Now we have our layer2 decoded code that we can regex out the values we need.

```
                matches =
re.findall('''\xb9.\x00\x00\x00\xf3\xa4\x83\xc6.\x83\xe8''', temp)
                if len(matches) > 0:
                        (dc, chunk_length, dc, dc, dc, addval) =
struct.unpack_from('<BBIBHB',matches[0])
                        out = ""
                        total_block = chunk_length + addval
```

So we can now fixup our other blob of data by copying over the chunks.

```
                        i = 0
                        while i < len(blob2):
                                out += blob2[i:i+chunk_length]
                                i += total_block
                        out2 = bytearray(out)
```

Then we just XOR using the same key and check if we need to decompress.

```
                        for i in range(len(out2)):
                                out2[i] ^= key[i%len(key)]
                        if 'MZ' != out2[:2]:
                                out3 = lznt_p.decompress_data(out2)
                        else:
                                out3 = out2
                        open(sys.argv[1]+'_embedded.bin', 'wb').write(out3)
                else:
                        print("Unknown layer2!")
```

Writing public unpackers basically is doing QA or quality assurance for the guys writing the packers but sometimes it's good to do to prove or disprove a theory such as the one I stated above about this layer being a packer layer versus related to Plurox. It's also good because it can prove beneficial to aspiring malware researchers out there, it's usually easier to learn through mimicry so hopefully the above is useful to someone out there.

Now after unpacking a few samples I noticed a few interesting things:

- There's more than one domain used by this
- This is indeed a packer used by more than just Plurox

I only did a few samples but I found the following domains being leveraged:

```
google-public-dns-b.google.com
google-public-dns-a.google.com
example.com
```

I also pretty quickly found a sample that unpacks to a DarkComet sample:

```
MD5     7e12e4b19e000e30385fc995db4fe837
SHA-1   2dacc210e01f380765c7b9fe0dcf7f650f98bbde
SHA-256         e0bdab9458543ac59ce6030e3b66dd503c2c35c04596eb3e9e30188223946155

[+] Printing Config to screen
   [-] Key: CHANGEDATE   Value: 0
   [-] Key: COMBOPATH    Value: 3
   [-] Key: DIRATTRIB    Value: 0
   [-] Key: EDTDATE      Value: 16/04/2007
   [-] Key: EDTPATH      Value: Soft\olp32.exe
   [-] Key: FILEATTRIB   Value: 0
   [-] Key: FTPHOST      Value:
   [-] Key: FTPPASS      Value:
   [-] Key: FTPPORT      Value:
   [-] Key: FTPROOT      Value:
   [-] Key: FTPSIZE      Value:
   [-] Key: FTPUPLOADK   Value:
   [-] Key: FTPUSER      Value:
   [-] Key: FWB  Value: 0
   [-] Key: GENCODE      Value: 8Ub1461JKvo2
   [-] Key: INSTALL      Value: 1
   [-] Key: KEYNAME      Value: OLP Software
   [-] Key: MELT         Value: 0
   [-] Key: MUTEX        Value: CMQCKTN
   [-] Key: NETDATA      Value: 185.146.157.143:1604
   [-] Key: OFFLINEK     Value: 1
   [-] Key: PERSINST     Value: 0
   [-] Key: PWD  Value:
   [-] Key: SID  Value: VM
[+] End of Config
```

Some other samples using the protection layer:

Azorult:

```
MD5     2cf6634a78c734876377d13f7cd4c178
SHA-1   d63616dd6e69218c709fffce76834406ab52e6f6
SHA-256         6e70a71063acdd9570fea8698d090d87e4767f80a643e121839b4449924f2d8c
```

Baldr:

```
MD5     21661041c0912c97cbc9f1e16e5f5d06
SHA-1   19117be8e15e08acba72a1d7c732bf2e87cb4992
SHA-256         f1ea3330bf0b5bf426328d41b0faae689366070b4013e92fe87cc1de55eba2c6
```

Clipboard crypto wallet replacer

```
MD5     ea3a524f3375232661bbee54367d92ba
SHA-1   9a000e8d1b11c7d35af1f54c626e812b41ab0e64
SHA-256         bdde9b0ca484ca08e0572b846f2a7ba989d999898ce1c095d0b4b678993b8d28
```

References:

1. https://github.com/erocarrera/pefile
2. https://github.com/google/rekall/blob/master/rekall-core/rekall/plugins/filesystems/lznt1.py
3. https://github.com/rapid7/metasploit-framework
4. https://github.com/kevthehermit/RATDecoders