

IMT4116_S17 Exam

Matthías Ragnarsson

June 5, 2017

Basic Static Analysis

a.

The IoC's are:

- 50/61 detection ratio on VirusTotal with names mostly including Trojan.
- Contains executable paths
 - `http://qrojectonline.ath.cx/yourserver.exe`
 - `http://qrojectonline.ath.cx/yourserver2.exe`
- Contains following suspicious API calls
 - `InternetReadFile`
 - `WinExec`
- Contains old browser type, probably for internet traffic
 - `/Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)`

The hypothesis is that this is a trojan that downloads `yourserver.exe` or `yourserver2.exe` to install a backdoor, i.e. server on the victim machine.

b.

The IoC's are:

- 56/62 detection rate on VirusTotal with names mostly including Trojan and Backdoor.
- A comment by PayloadSecurity on VirusTotal links to a report¹ with a risk assessment indicating the abilities of remote access, persistence, host fingerprinting. It connects to `irc.mcs.net` or `104.199.126.136`. It also informs of unusually high entropy sections, monitoring of registry changes and creation of auto-execution registry entries.

¹<https://www.hybrid-analysis.com/sample/9de606047ae141a872a7ddb78782fc8a8da5518e879b2239ec931560b7983ba8?environmentId=100>

- Finding of `SOFTWARE\MICROSOFT\WINDOWS\CurrentVersion\Run` in strings, indicating supporting persistence mechanism.

Hypothesis is then that it is a trojan backdoor.

c.

- The sample is listed on VirusTotal with 32/59 detection rating with names mostly including Trojan and Backdoor.
- UPX packer was detected on VirusTotal, which is also apparent from two section names UPX0, UPX1 and UPX3.
- `urlmon.URLDownloadToFile` and `SHELL32.ShellExecuteA` indicate a download and execution of a file.
- IRC commands among the strings.

With the aforementioned clues, I hypothesize this to be a dropper, utilizing IRC.

Basic Dynamic Analysis

a.

There is a persistence mechanism for the file is put into the registry as shown below. With Procmon and CaptureBAT, I didn't see any network activity or creation of files or other value setting registry operations.

Time of Day	Process Name	PID	Operation	Path	Result	Detail
8:06:25.7569654 PM	exam_2a.exe	2764	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Run	SUCCESS	Desired Access: Set Value, Disposition: REG_OPENED_EXISTING_KEY
8:06:25.7570060 PM	exam_2a.exe	2764	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Run(Default)	SUCCESS	Type: REG_SZ, Length: 78, Data: C:\Users\rev_eng\Downloads\exam_2a.exe

Also it tries to contact the IP address 104.229.85.209 as shown below by fakedns. It was also shown in Process Hacker 2.

```

Respuesta: www.download.windowsupdate.com. -> 192.168.56.103
Respuesta: teredo.ipv6.microsoft.com. -> 192.168.56.103
Respuesta: 104.229.85.209.in-addr.arpa. -> 192.168.56.103
jRespuesta: teredo.ipv6.microsoft.com. -> 192.168.56.103

```

I hypothesize that this is some backdoor.

b.

Justing from the Procmon logs, the file activity shows accesses to browser history and cookies as well as access to "cryptbase.dll" and performance of network activity, i.e. contacting 146.27.234.1 as shown with fakedns below (also seen in Process Hacker 2). The process did not stay very long in

execution. My first hypothesis is that it is a cookie and browsing history stealer. The supporting Procmon log parts are shown below.

2:40:47.3525024 PM	exam_2b.exe	3220	UDP Send	WIN-MVG8V3FP36.52981 -> WIN-MVG8V3FP36.52981	SUCCESS	Length: 1, seqnum: 0, connid: 0
2:40:47.3525471 PM	exam_2b.exe	3220	UDP Receive	WIN-MVG8V3FP36.52981 -> WIN-MVG8V3FP36.52981	SUCCESS	Length: 1, seqnum: 0, connid: 0
2:40:47.2100840 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\index.dat		
2:40:47.2103486 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\index.dat		
2:40:47.2113820 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Roaming\Microsoft\Windows\Cookies		
2:40:47.2116326 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Roaming\Microsoft\Windows\Cookies		
2:40:47.2119340 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Roaming\Microsoft\Windows\Cookies\index.dat		
2:40:47.2121924 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Roaming\Microsoft\Windows\Cookies\index.dat		
2:40:47.2135588 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\History\History.IE5		
2:40:47.2137633 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\History\History.IE5		
2:40:47.2140088 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\History\History.IE5\index.dat		
2:40:47.2142720 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\History\History.IE5\index.dat		
2:40:47.2152146 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5		
2:40:47.2155618 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\desktop.ini		
2:40:47.2159161 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\History\History.IE5		
2:40:47.2162619 PM	exam_2b.exe	3220	CreateFile	C:\Users\rev_eng\AppData\Local\Microsoft\Windows\History\History.IE5\desktop.ini		

```

Respuesta: teredo.ipv6.microsoft.com. -> 192.168.56.103
Respuesta: 146.27.234.1.in-addr.arpa. -> 192.168.56.103
Respuesta: teredo.ipv6.microsoft.com. -> 192.168.56.103

```

Advanced Analysis

a. Basic Functionality

i.

8, seen via number of dotted arrows in the assembly listing.

ii.

It jumps on condition of the `eax` register not being 0, as determined by The Zero Flag which is set by the previous `dec` instruction if `eax` is 0 after the decrement.

iii.

It jumps on condition of the `esi` register being less than or equal to 0x100 or 256.

iv.

It jumps if the Zero Flag is set by the previous `cmp` instruction. Therefore it jumps if `[ebp + var_9]` is equal to 0. `[ebp + var_9]` is some local variable of the containing function at byte offset `ebp - 9`.

b.

i.

Checking the imports with rabin2, we see the functions **GetKeyState** and **GetAsyncKeyState** which check for the key state according to their documentation:

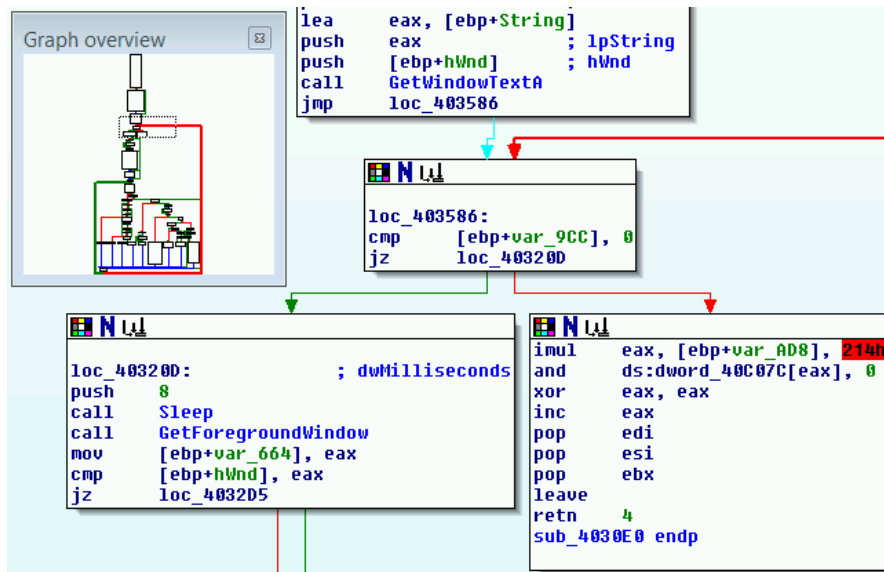
```
(~/exam)
❯ rabin2 -i exam_3b.exe | grep Key
Warning: check (Var szKey)
Warning: bad parsing Var
Warning: bad parsing (VS_VERSIONINFO VarFileInfo)
ordinal=003 plt=0x00414428 bind=NONE type=FUNC name=USER32.DLL_GetKeyState
ordinal=004 plt=0x0041442c bind=NONE type=FUNC name=USER32.DLL_GetAsyncKeyState
ordinal=005 plt=0x00414430 bind=NONE type=FUNC name=USER32.DLL_MapVirtualKeyA
ordinal=002 plt=0x00414450 bind=NONE type=FUNC name=ADVAPI32.DLL_RegCreateKeyA
ordinal=003 plt=0x00414454 bind=NONE type=FUNC name=ADVAPI32.DLL_RegCreateKeyExA
ordinal=004 plt=0x00414458 bind=NONE type=FUNC name=ADVAPI32.DLL_RegCloseKey
ordinal=005 plt=0x0041445c bind=NONE type=FUNC name=ADVAPI32.DLL_RegOpenKeyA
```

Checking the cross-references or calls to those functions with radare2, we obtain the addresses 0x4032db, 0x403306, 0x403332 and 0x4032f3:

```
[0x004011cb]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[0x004011cb]> axt sub.USER32.DLL_GetKeyState_d98
call 0x4032db call sub.USER32.DLL_GetKeyState_d98 in unknown function
call 0x403306 call sub.USER32.DLL_GetKeyState_d98 in unknown function
call 0x403332 call sub.USER32.DLL_GetKeyState_d98 in unknown function
[0x004011cb]> axt sub.USER32.DLL_GetAsyncKeyState_da4
call 0x4032f3 call sub.USER32.DLL_GetAsyncKeyState_da4 in unknown function
[0x004011cb]> █
```

ii.

As is shown in the graph overview in the image below, there is one big loop in which the calls to **GetKeyState** and **GetAsyncKeyState** are called. Furthermore, the image shows the first basic block that enters the loop (the top box) and the last basic block of the loop (the middle box) that conditionally exits the loop. The beginning of the loop or the address of the jump into it is 0x00403208 and the end or the address of the jump out of the loop is 0x0040358D.



iii.

The number is located in variable $[ebp + var_9CC]$ in the containing polling function. The variable is obtained from a 4096 byte array which is read from 0x412e10 in the .data section to $[ebp + var_408]$. The dword which is consequently written into $[ebp + var_9CC]$ lies at address $0x41284C + (0x9cc - 0x408) = 0x412e10$. Further analysis is needed to determine this value.

c.

i.

Checking the hex dump as shown below, reveals a high prevalence of 0x13 bytes and unusually few 0x0 bytes. This makes single byte XOR encoding seem likely since in that case all 0x0 bytes transform to the single byte key as value.

```

[0x00000000]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00000000 5e49 8313 1013 1313 1713 1313 ecec 1313 ^I.....
0x00000010 ab13 1313 1313 1313 5313 1313 1313 1313 .....S.....
0x00000020 1313 1313 1313 1313 1313 1313 1313 1313 .....
0x00000030 1313 1313 1313 1313 1313 1313 1312 1313 .....
0x00000040 1d0c a91d 13a7 1ade 32ab 125f de32 477b .....2..._..2G{
0x00000050 7a60 3363 617c 7461 727e 3370 727d 7d7c z`3ca|tar~3pr}}|
0x00000060 6733 7176 3361 667d 337a 7d33 575c 4033 g3qv3af}3z}3W\@3
0x00000070 7e7c 7776 3d1e 1e19 3713 1313 1313 1313 ~|wv=...7.....
0x00000080 43a2 f5bb 07c3 9be8 07c3 9be8 07c3 9be8 C.....
0x00000090 0ebb 08e8 0dc3 9be8 3c9d 98e9 06c3 9be8 .....<.....
0x000000a0 3c9d 9ee9 15c3 9be8 3c9d 9fe9 0ac3 9be8 <.....<.....
0x000000b0 3c9d 9ae9 03c3 9be8 da3c 50e8 04c3 9be8 <.....<P.....
0x000000c0 07c3 9ae8 38c3 9be8 909d 92e9 05c3 9be8 ....8.....
0x000000d0 959d 64e8 06c3 9be8 909d 99e9 06c3 9be8 ..d.....
0x000000e0 417a 707b 07c3 9be8 1313 1313 1313 1313 Azp{.....
0x000000f0 1313 1313 1313 1313 1313 1313 1313 .....
[0x00000000]>

```

ii.

Trying 0x13 as a single byte XOR key on the whole file reveals proper PE executable format as shown below, revealing it to indeed be the key. The meaning of the decoding command is to XOR 0x13 at (@) current location (\$\$), which is the first byte of the file, and throughout the byte length of the file (\$s).

```

[0x00000000]> wox 0x13 @ $$!$s
[0x00000000]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00000000 4d5a 9000 0300 0000 0400 0000 ffff 0000 MZ.....
0x00000010 b800 0000 0000 0000 4000 0000 0000 0000 .....@.....
0x00000020 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x00000030 0000 0000 0000 0000 0000 0000 0001 0000 .....
0x00000040 0e1f ba0e 00b4 09cd 21b8 014c cd21 5468 .....!..L.!Th
0x00000050 6973 2070 726f 6772 616d 2063 616e 6e6f is program canno
0x00000060 7420 6265 2072 756e 2069 6e20 444f 5320 t be run in DOS
0x00000070 6d6f 6465 2e0d 0d0a 2400 0000 0000 0000 mode....$.
0x00000080 50b1 e6a8 14d0 88fb 14d0 88fb 14d0 88fb P.....
0x00000090 1da8 1bfb 1ed0 88fb 2f8e 8bfa 15d0 88fb ...../.....
0x000000a0 2f8e 8dfa 06d0 88fb 2f8e 8cfa 19d0 88fb /...../.....
0x000000b0 2f8e 89fa 10d0 88fb c92f 43fb 17d0 88fb /...../C.....
0x000000c0 14d0 89fb 2bd0 88fb 838e 81fa 16d0 88fb ....+.....
0x000000d0 868e 77fb 15d0 88fb 838e 8afa 15d0 88fb ..w.....
0x000000e0 5269 6368 14d0 88fb 0000 0000 0000 0000 Rich.....
0x000000f0 0000 0000 0000 0000 0000 0000 0000 .....

```

iii.

The file was decoded as seen in the image above.

iv.

The screenshots below show this information. The first one is from the peframe tool and the second one from rabin2 revealing the section names.

```
File type      PE32 executable (console) Intel 80386, for MS Windows
File name      exam_3c.exe
File size      11264
Hash MD5       eafb326e8f3f278be9e5b534d4238218
Compile time   2017-05-31 17:53:59
```

```
(~/exam)
$ rabin2 -S exam_3c.exe
[Sections]
idx=00 vaddr=0x00401000 paddr=0x00000400 sz=4096 vsz=4096 perm=m-r-x name=.text
idx=01 vaddr=0x00402000 paddr=0x00001400 sz=4096 vsz=4096 perm=m-r-- name=.rdata
idx=02 vaddr=0x00403000 paddr=0x00002400 sz=512 vsz=4096 perm=m-rw- name=.data
idx=03 vaddr=0x00404000 paddr=0x00002600 sz=512 vsz=4096 perm=m-r-- name=.gfids
idx=04 vaddr=0x00405000 paddr=0x00002800 sz=512 vsz=4096 perm=m-r-- name=.rsrc
idx=05 vaddr=0x00406000 paddr=0x00002a00 sz=512 vsz=4096 perm=m-r-- name=.reloc
6 sections
```

d.

i.

The string size being a multiple of 4 with no "=" at the end and being alphanumeric points to it being a base64 encoding, which it is (see ii.).

ii.

The decoding is shown in the image below.

```
$ echo "Q29uZ3JhdHVzYXRpb25zLCB5b3Ugc3VjY2Vzc2Z1bGx5IGRlY29kZWQgdGhpcyBtZXNz\
yb2NlZWQh" | base64 -d -
Congratulations, you successfully decoded this message, you may proceed!
```

e.

i.

Trying to run the sample for network capture didn't work, seeing as I didn't capture any traffic running it. So I contend with static analysis. By running strings, we find one IP address 81.95.152.178. To see whether it is indeed used to connect, we can see the network function (InternetConnectA) taking it as argument as shown below in the function referencing the string.

```

[0x00401cd6]> axt str.81.95.152.178
data 0x40144b push str.81.95.152.178 in sub.WININET.dll_InternetOpenA_400
[0x00401cd6]> s 0x40144b
[0x0040144b]> pd 10
|      0x0040144b      6830784000      push str.81.95.152.178      ; 0x407830 ; "81.95.15
|      0x00401450      50              push eax
|      0x00401451      ff15c8604000    call dword [sym.imp.WININET.dll_InternetConnectA]
|      0x00401457      3bc3           cmp eax, ebx
|      0x00401459      a3ec8a4000     mov dword [0x408aec], eax ; [0x408aec:4]=0
|      0x0040145e      746b          je 0x4014cb
|      0x00401460      53            push ebx
|      0x00401461      6800000804     push 0x4080000
|      0x00401466      53            push ebx
|      0x00401467      6830744000     push str.fe25de0be9887dbb2ac25dad792fce2a ; 0x4074
[0x0040144b]>

```

ii.

The HTTP protocol is used as evidenced below.

iii.

After using the IP in establishing connection as explained in (i.), the same function then creates an HTTP request handle using that connection as shown below. The connection handle is carried in `eax` and passed to the `HTTPOpenRequestA` API call at address `0x00401476` which creates the read handle. The read handle is stored in `0x408af0` and is then later passed to `InternetReadFile` API call at address `0x004014ea` which reads the returned data.

```

0x00401451      ff15c8604000    call dword [sym.imp.WININET.dll_InternetConnectA]
0x00401457      3bc3           cmp eax, ebx
0x00401459      a3ec8a4000     mov dword [0x408aec], eax ; [0x408aec:4]=0
0x0040145e      746b          je 0x4014cb
0x00401460      53            push ebx
0x00401461      6800000804     push 0x4080000
0x00401466      53            push ebx
0x00401467      6830744000     push str.fe25de0be9887dbb2ac25dad792fce2a ; 0x4074
0x0040146c      53            push ebx
0x0040146d      ff7508         push dword [ebp + 8]
0x00401470      681c824000     push str.POST ; 0x40821c ; "POST"
0x00401475      50            push eax
0x00401476      ff15cc604000    call dword [sym.imp.WININET.dll_HttpOpenRequestA]
0x0040147c      3bc3           cmp eax, ebx
0x0040147e      a3f08a4000     mov dword [0x408af0], eax ; [0x408af0:4]=0

```

```

0x004014df      50            push eax
0x004014e0      ff751c         push dword [ebp + 0x1c]
0x004014e3      56            push esi
0x004014e4      ff35f08a4000    push dword [0x408af0]
0x004014ea      ff15bc604000    call dword [sym.imp.WININET.dll_InternetReadFile]

```

iv.

Running strings on the executable, we see 6 strings that look like promising command candidates: `wait`, `execute`, `icmp_flood`, `http_flood`, `download` and `sysinfo`. Upon further inspection of the function

cross referencing the sysinfo string, they seem very likely to be commands indeed. I give only the summary of the function below, showing its strings and calls. The candidate commands are passed as strings to the function fcn.00401b20. The return value prompts a jump past the other calls, indicating some kind of if-else construct. The command seems to come from the index.php file fetched in the beginning.

```
[0x0040144b]> axt @ str.sysinfo
data 0x4015cd push str.sysinfo in sub.urlmon.dll_URLDownloadToFileA_57b
[0x0040144b]> pdsf @ sub.urlmon.dll_URLDownloadToFileA_57b
0x00401598 str._index.php
0x0040159d call sub.WININET.dll_InternetOpenA_400
0x004015b6 call fcn.00401c3a
0x004015bc call fcn.00401c2f
0x004015c6 call fcn.00401c3a
0x004015cd str.sysinfo
0x004015d3 call fcn.00401b20
0x004015e2 call sub.KERNEL32.dll_GlobalMemoryStatus_0
0x004015f0 str.download
0x004015f6 call fcn.00401b20
0x00401603 call fcn.00401c3a
0x0040160c call fcn.00401c3a
0x00401616 call fcn.00401188
0x00401620 str.http_flood
0x00401626 call fcn.00401b20
0x00401633 call fcn.00401c3a
0x0040163c call fcn.00401c3a
0x00401642 call fcn.00401c2f
0x0040164c call fcn.00401c3a
0x00401656 call fcn.00401c3a
0x0040165c call fcn.00401c2f
0x00401669 call sub.WININET.dll_InternetOpenA_20d
0x00401676 str.icmp_flood
0x0040167c call fcn.00401b20
0x00401689 call fcn.00401c3a
0x00401692 call fcn.00401c3a
0x00401698 call fcn.00401c2f
0x0040169f call fcn.004012b6
0x004016a9 str.execute
0x004016af call fcn.00401b20
0x004016bc call fcn.00401c3a
0x004016c5 call sub.KERNEL32.dll_CreateProcessA_32d
0x004016d2 str.wait
0x004016d8 call fcn.00401b20
0x004016e5 call fcn.00401c3a
0x004016eb call fcn.00401c2f
0x004016f4 call dword [sym.imp.KERNEL32.dll_Sleep]
0x00401707 call fcn.00401b20
0x00401717 call fcn.00401a40
[0x0040144b]>
```

f.

i.

Running strings on the executable, we see the VBoxService.exe string, a hallmark of the Virtualbox guest services (probably in an earlier version as I couldn't verify it in my own VM). On checking where the string is cross referenced, it is passed to a function as seen in the screenshots below. The latter screenshot shows the loop within the function that iterates the list of running processes and compares it to the string. If a match is found, the function returns that result. The address of the check can be said to be the Kernel32.dll strcmp function at address 0x00408a95.

```
[0x00408a28]> axt @ str.VBoxService.exe
data 0x408b0f mov dword [esp], str.VBoxService.exe in fcn.00408b09
[0x00408a28]> pdf @ fcn.00408b09
/ (fcn) fcn.00408b09 20
|   fcn.00408b09 ();
|       ; CALL XREF from 0x00408c60 (fcn.00408c47)
|       0x00408b09      55          push ebp
|       0x00408b0a      89e5        mov ebp, esp
|       0x00408b0c      83ec18      sub esp, 0x18
|       0x00408b0f      c70424eb5a41. mov dword [esp], str.VBoxService.exe
|       0x00408b16      e80dffffff  call fcn.00408a28
|       0x00408b1b      c9          leave
|       0x00408b1c      c3          ret
```

```
0x00408a73      e818ab0000  call sub.KERNEL32.DLL_Process32First_590
0x00408a78      83ec08      sub esp, 8
0x00408a7b      85c0        test eax, eax
,=< 0x00408a7d      7436        je 0x408ab5
| 0x00408a7f      8db5e4fefff lea esi, [local_11ch]
| 0x00408a85      8dbdc0fefff lea edi, [local_140h]
|       ; JMP XREF from 0x00408ab3 (fcn.00408a28)
.---> 0x00408a8b      8b4508      mov eax, dword [arg_8h] ; [0x8:4]=4
|| 0x00408a8e      89442404    mov dword [esp + 4], eax
|| 0x00408a92      893424      mov dword [esp], esi
|| 0x00408a95      e8bea90000  call sub.msvcrt.dll_strcmp_458
|| 0x00408a9a      85c0        test eax, eax
===< 0x00408a9c      7504        jne 0x408aa2
|| 0x00408a9e      b001        mov al, 1
===< 0x00408aa0      eb15        jmp 0x408ab7
!|       ; JMP XREF from 0x00408a9c (fcn.00408a28)
.---> 0x00408aa2      897c2404    mov dword [esp + 4], edi
|| 0x00408aa6      891c24      mov dword [esp], ebx
|| 0x00408aa9      e8eaaa0000  call sub.KERNEL32.DLL_Process32Next_598
|| 0x00408aae      83ec08      sub esp, 8
|| 0x00408ab1      85c0        test eax, eax
`==< 0x00408ab3      75d6        jne 0x408a8b
```

ii.

We see that the function fcn.00408b09 above is cross referenced from fcn.00408c47. Examining that function, we see many other checking functions similar to fcn.00408b09. A summary of the

strings and functions in `fcn.00408c47` are shown below. We see the sample additionally checks for the automated analysis tool Joebox, Sandboxie and Wireshark. It additionally tries to crash OllyDbg 1.1 with the `OutputDebugString` vulnerability. It also tests for any debugger in general with `IsDebuggerPresent` and a timing attack with `GetTickCount`. It also checks for the presence of some Window, but I'm not sure what yet.

[illegible]

iii.

From our findings in (ii.) and examining the function, it is clear the function check for a laboratory environment and returns a boolean true if so, otherwise false.

iv.

From 0x00408cc3 as shown below.

```
[0x00408c47]> axt @ fcn.00408c47
call 0x408cc3 call fcn.00408c47 in unknown function
```

v.

If the laboratory environment is detected, then the malware terminates, otherwise it continues on (for some routine checks before continuing). This is shown below.

```
[0x00408cc3]> pd 15
; JMP XREF from 0x00408cf9 (fcn.00408c47 + 178)
.-> 0x00408cc3 e87fffffff call fcn.00408c47
| 0x00408cc8 84c0 test al, al
,=< 0x00408cca 7417 je 0x408ce3
|| 0x00408ccc c74424040000. mov dword [esp + 4], 0
|| 0x00408cd4 c70424ffffff. mov dword [esp], 0xffffffff ; [0xffffffff:4
|| 0x00408cdb e840a80000 call sub.KERNEL32.DLL_TerminateProcess_520
|| 0x00408ce0 83ec08 sub esp, 8
!| ; JMP XREF from 0x00408cca (fcn.00408c47 + 131)
^--> 0x00408ce3 c70424010000. mov dword [esp], 1
| 0x00408cea e811a80000 call sub.KERNEL32.DLL_Sleep_500
| 0x00408cef 83ec04 sub esp, 4
| 0x00408cf2 803d7a9e4100. cmp byte [0x419e7a], 0
^< 0x00408cf9 74c8 je 0x408cc3
; JMP XREF from 0x00408cc1 (fcn.00408c47 + 122)
0x00408cfb b801000000 mov eax, 1
0x00408d00 c9 leave
0x00408d01 c20400 ret 4
```

g.

i.

The most likely reason for a mutex is to avoid duplicate instances of the same malware by checking if another mutex with the same name is running before creating a new one.

ii.

iii.

It is address 0x0040465c as shown below.

```

| 0x00404647 6880094100 push str.TreasureHunter_version_0.1_Alpha__created_by_Jo
version 0.1 Alpha, created by Jolly Roger (jollyroger@prv.name) for BearsInc. Greets to Xy
| 0x0040464c ff1584c04000 call dword [sym.imp.KERNEL32.dll_OutputDebugStringW] ; 0
| ; JMP XREF from 0x00404635 (main)
-> 0x00404652 ff35a04e4100 push dword [0x414ea0]
0x00404658 6a00 push 0
0x0040465a 6a00 push 0
0x0040465c ff1550c04000 call dword [sym.imp.KERNEL32.dll_CreateMutexA] ; 0x40c05
0x00404662 8bd8 mov ebx, eax
0x00404664 8d4601 lea eax, [esi + 1] ; "Z."
| ; JMP XREF from 0x0040466c (main)
-> 0x00404667 8a0e mov cl, byte [esi]
| 0x00404669 46 inc esi
| 0x0040466a 84c9 test cl, cl
| 0x0040466c 75f9 jne 0x404667
| 0x0040466e 2bf0 sub esi, eax

```

iv.

It takes 3 inputs as such `CreateMutexA(0, 0, n)` where `n` is a pointer to the string of the name which in this case is at memory location `0x414ea0` as shown above. These arguments are pushed on the stack from in order of `n` to `0`. We see below that the origins of `n` is from a call to the function `fcn.00402380` (which returns a value into `eax` first).

```

0x004045f3 83e4f8 and esp, 0xffffffff
0x004045f6 83ec24 sub esp, 0x24 ; '$'
0x004045f9 a100204100 mov eax, dword [0x412000] ; section..data ;
0x004045fe 33c4 xor eax, esp
0x00404600 89442420 mov dword [local_20h], eax
0x00404604 53 push ebx
0x00404605 56 push esi
0x00404606 8b7510 mov esi, dword [arg_10h] ; [0x10:4]=184
0x00404609 57 push edi
0x0040460a 68c04e4100 push 0x414ec0
0x0040460f 89742414 mov dword [local_14h], esi
0x00404613 ff157cc04000 call dword [sym.imp.KERNEL32.dll_InitializeCr
0x00404619 e862ddffff call fcn.00402380
0x0040461e a3a04e4100 mov dword [0x414ea0], eax ; [0x414ea0:4]=0

```

v.

You get it from subroutine `fcn.00402380` as explained in (iv.). I'm not sure yet what to say on its origins.

Combined Analysis

a.

`exam_4a.exe` is the same as `exam_1b.exe`, so I refer to the Basic Static Analysis section b regarding `exam_4a.exe`. `exam_4b.exe` is an ASCII text file. IoCs regarding `exam_4b.exe` is the following:

- VirusTotal claims that `exam_4b.exe` is a part of a larger zip file. Its contents are `svrvc.exe` (which I believe is the same as `exam_1a.exe`) and `gus.ini`. I don't see the hash of `gus.ini`, but it

is the same size as exam_4b.exe. I'll therefore assume they're the same file as well. Similar to the exam_4a.exe, the larger zip file has a 45/55 detection ratio and is variously named trojan or backdoor.

b.

It tries to set the following registry key below, but without success, perhaps requiring higher privileges.

RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	Desired Access: All Access
RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Service Profiler	ACCESS DENIED	Type: REG_SZ, Length: 20, Data: srvcpx.exe
RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	

It also tries to read "gus.ini" as shown below.

CreateFile	C:\Windows\System32\gus.ini	NAME NOT FOUND	Desired Access: Generic Read
CreateFile	C:\Windows\System32\gus.ini	NAME NOT FOUND	Desired Access: Generic Read
CreateFile	C:\Windows\System32\gus.ini	NAME NOT FOUND	Desired Access: Generic Read
CreateFile	C:\Windows\System32\gus.ini	NAME NOT FOUND	Desired Access: Generic Read
CreateFile	C:\Windows\System32\gus.ini	NAME NOT FOUND	Desired Access: Generic Read
CreateFile	C:\Windows\System32\gus.ini	NAME NOT FOUND	Desired Access: Generic Read
CreateFile	C:\Windows\System32\gus.ini	NAME NOT FOUND	Desired Access: Generic Read
CreateFile	C:\Windows\System32\gus.ini	NAME NOT FOUND	Desired Access: Generic Read

There was some network activity noticed for exam_4a.exe in Process Hacker 2. It tried to connect to an IRC server on port 6667.

c.

Yes, now the sample successfully writes to the same registry key mentioned above.

RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	Desired Access: All Access
RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Service Profiler	SUCCESS	Type: REG_SZ, Length: 20, Data: srvcpx.exe
RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	

d.

In my dynamic analysis environment, I set up an IRC service in Inetsim and ran Tshark to capture incoming packets. Reading the pcapng file revealed the following. We see the nickname is "mikey" and channel "#daFuck".

```

Internet Relay Chat
Request: NICK :mikey
Command: NICK
Trailer: mikey
Request: USER Yp Yp Yp :fight me, pussy
Command: USER
Command parameters
Parameter: Yp
Parameter: Yp
Parameter: Yp
Trailer: fight me, pussy
Request: JOIN #daFuck
Command: JOIN
Command parameters
Parameter: #daFuck

```

e.

The difference is that it sends traffic now to port 6666 instead of 6667. There is no IRC communication, but just TCP packets are sent. Still the TCP packet contents seem to contain much the same IRC messages as before. It seems to repeat trying to change its nickname, which is already taken. I don't have time to look into that deeper.

f.

Debugging the sample with x64dbg as shown below reveals the function to be a string decoder for various strings. Two of the decoded strings are "gus.ini" and "mikey".

00401418	F3 A4	rep movsb byte ptr es:[edi],byte ptr ds	
0040141D	68 34 80 00	push exam_4a.408034	408034:"gus.ini"
00401422	E8 9F FE FF	call exam_4a.4012C6	
00401427	68 3C 80 00	push exam_4a.40803C	40803C:"mikey"
0040142C	E8 95 FE FF	call exam_4a.4012C6	
00401431	68 42 80 00	push exam_4a.408042	408042:"setpr"
00401436	E8 8B FE FF	call exam_4a.4012C6	
0040143B	68 48 80 00	push exam_4a.408048	408048:"jiggy"
00401440	E8 81 FE FF	call exam_4a.4012C6	
00401445	68 4E 80 00	push exam_4a.40804E	40804E:"daFuck"
0040144A	E8 77 FE FF	call exam_4a.4012C6	
0040144F	68 53 80 00	push exam_4a.408053	408053:"daFuckwhat"
00401454	E8 6D FE FF	call exam_4a.4012C6	
00401459	68 60 80 00	push exam_4a.408060	408060:"fight me, pussy"
0040145E	E8 63 FE FF	call exam_4a.4012C6	
00401463	68 70 80 00	push exam_4a.408070	408070:"irc.mcs.net:6667"
00401468	E8 59 FE FF	call exam_4a.4012C6	
0040146D	68 81 80 00	push exam_4a.408081	408081:"AGGRESSIVE"
00401472	E8 4F FE FF	call exam_4a.4012C6	
00401477	68 8C 80 00	push exam_4a.40808C	40808C:"ISON a b c d e f g h i j k
0040147C	E8 45 FE FF	call exam_4a.4012C6	
00401481	68 96 00 00	push 96	
00401486	68 00 64 00	push exam_4a.406400	

g.

i.

Using radare2, it is revealed to be at address 0x40366a as is seen in the first line of the last command shown below.

```
[0x004011cb]> f | grep fopen
0x0000a4ce 4 reloc.CRTDLL.DLL_fopen_206
0x0040a290 0 sym.imp.CRTDLL.DLL_fopen
0x0040573c 6 sub.CRTDLL.DLL_fopen_73c
[0x004011cb]> axt @ sub.CRTDLL.DLL_fopen_73c
call 0x40366a call sub.CRTDLL.DLL_fopen_73c in fcn.0040363d
call 0x4038f2 call sub.CRTDLL.DLL_fopen_73c in fcn.004038bb
call 0x4041a5 call sub.CRTDLL.DLL_fopen_73c in fcn.00404195
call 0x404390 call sub.CRTDLL.DLL_fopen_73c in unknown function
[0x004011cb]>
```

ii.

Using radare2, we find the cross references to that function as shown below. I pick the last one at address 0x4036ba since it is in a known function which fopen was also in.

```
[0x004011cb]> axt @ fcn.00405366
call 0x401e7a call fcn.00405366 in unknown function
call 0x401f05 call fcn.00405366 in unknown function
call 0x4036ba call fcn.00405366 in fcn.0040363d
```

Looping over the function in x64dbg as shown below, we see the string eax points to after the call. Call 1, 9 and 10 are the only ones in which a (nonnull) string is returned, shown below in succession.

0040366A	E8 A7 1C 00 00	call exam_4a.405366	eax: "NICK=m1key"
0040366F	89 C7	mov edi, eax	
004036BA	E8 A7 1C 00 00	call exam_4a.405366	eax: "SERVER1=efnet.cs.hut.fi:6666"
004036BF	89 C7	mov edi, eax	
004036C4	E8 A7 1C 00 00	call exam_4a.405366	edi: "SERVER2=efnet.demon.co.uk:6666", eax: "SERVER2=efnet.demon.co.uk:6666"
004036C9	89 C7	mov edi, eax	

h.

My findings indicate this malware sample, exam_2a.exe as being a trojan backdoor, likely left behind by a dropper that also left exam_2b.exe behind as it is used by exam_2a.exe which does not drop it on its own. It communicates with the domains irc.mcs.net over port 6667 and efnet.cs.hut.fi and efnet.demon.co.uk over port 6666. Evidence points to possible remote control from these domains, which could be confirmed with more analysis time.

Open Source

a.

WannaCry is malware classified as ransomware. Its purpose is financial gain, to encrypt the contents of victim Windows computers and to demand a ransom in bitcoins in order to have the contents decrypted (usable again)².

b.

The malware spreads between infected machines by utilizing a Samba (SMB) vulnerability³ in Windows machines. SMB is used mainly for file sharing⁴ between computers.

c.

You can use an automated reputable WannaCry aware anti-malware solution for detection such as Malwarebytes here. You can also look for indicators matching the file attributes listed here here. If you find such files, you can try to match them to the YARA signature found here. There are some solutions to remove it, such as here.

d.

Follow the recommended steps for protection from the solutions section here.

²<https://www.us-cert.gov/ncas/alerts/TA17-132A>

³<https://technet.microsoft.com/en-us/library/security/ms17-010.aspx>

⁴https://en.wikipedia.org/wiki/Server_Message_Block