

## Lab 12-02

Analyze the malware found in the file Lab12-02.exe.

# File Analysis

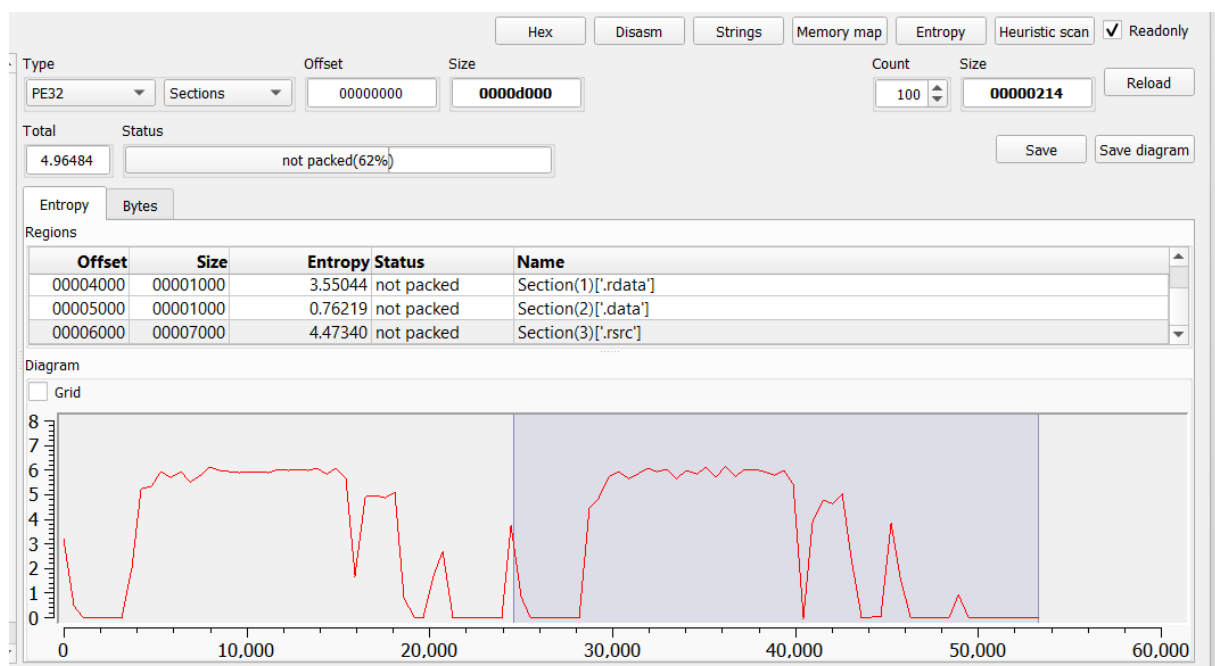
First of all, let's check what we are dealing with before execution.

**Lab12-02.exe** is a malware written using C++.

The file does not seem packed, entropy is not that high (almost 5), section names remain unchanged, virtual and physical sizes are almost the same.

When it comes to the topic of entropy, there are some suspicious peaks at resources section (highlighted area at image below), the 3 big peaks might tell us that there might be another file hiding there.

To confirm that, we have to take a look at it in further analysis.



Interesting flossed strings:

- \svchost.exe
- LOCALIZATION
- UNICODE

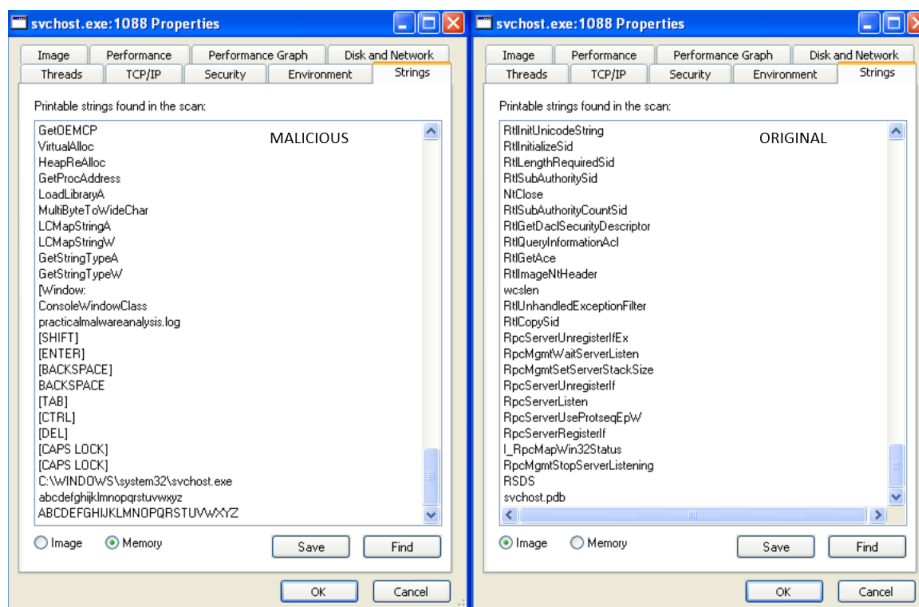
The malware declares only one library import (kernel32.dll), the functions hints us possible behavior of the sample:

- File operations
- Allocating memory, writing memory
- Thread management
- Process management
- Dynamically library loading
- Resources management

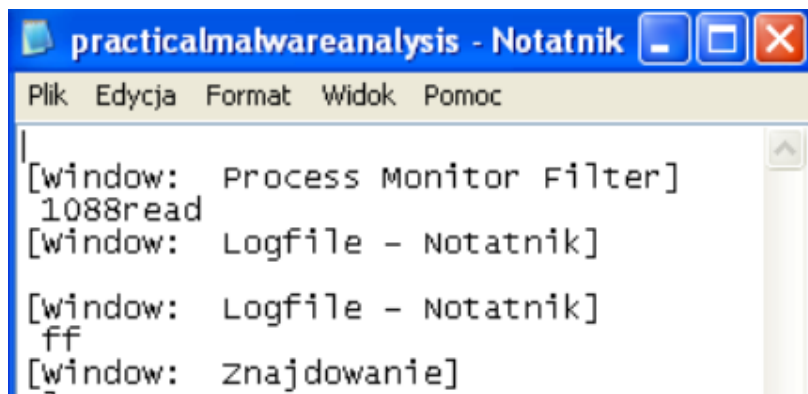
There is a section containing resources, inside it we have an entry “UNICODE/LOCALIZATION containing 24576 bytes of data. The data seems pretty random and does not reveal much at this point, for example there are three bigger loads of data (3504, 3263, 1928 bytes each) containing mostly 0x41, 0x40 values (corresponding A,B ASCII)

We will have to check at later steps how it resolves data after API calls to the resources.

In the svchost.exe memory strings there are some indicators that we are dealing with activity related to keylogging, there are bunch of keys such as [SHIFT], [ENTER], there is also a string path to the victim process and we have a possible name of a file storing gathered data **practicalmalwareanalysis.log**.



The .log file is being saved in the **Lab12-01.exe** directory, it saves the data about currently running window and it's keystrokes.

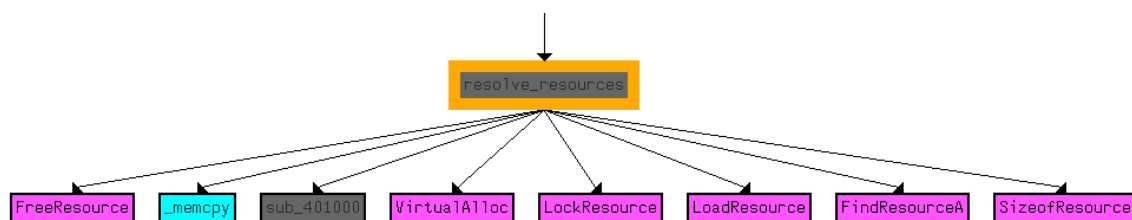


In the threads, there is no sign of any DLLs attached and the process was newly created after file execution, so it might process hollowing attack using suspension state.

Let's when we have some knowledge about it, let's put it now into disassembler to see what it does exactly.

The entry point for the executable is located at 0x4014E0, the code resolves system directory path using **GetSystemDirectoryA**, connects the string with [\\svchost.exe](#) to form a path to a victim file.

Later, at 0x401528 we have a call to a sub that I named **resolve\_resources**, as the name says, this function locates **UNICODE/LOCALIZATION** resource and allocates it in the memory of the currently running process for later usage.



After we gathered path to the victim, resolved the resources it calls another function that I called **start\_svchostWithResource**.

The function mainly focuses at implementing process hollowing. The attack divides to few steps: starting the victim process in a suspended state (dwCreationFlags set to 4), unmapping current victim memory using dynamically imported ntdll.dll function **NtUnmapViewOfSection**, allocating space in the victim memory (**VirtualAllocEx**), filling the space with the new code (**WriteProcessMemory**).

The writing call is used PE header and for each of the sections.

All that's left is to set the register by **SetThreadContext**, and execute the code with **ResumeThread** call.

```
.text:00401269
.text:00401269 loc_401269:
.text:00401269 mov     ecx, [ebp+var_8]
.text:0040126C xor     edx, edx
.text:0040126E mov     dx, [ecx+6]
.text:00401272 cmp     [ebp+var_70], edx
.text:00401275 jge     short loc_4012B9

.text:00401277 mov     eax, [ebp+var_4]
.text:0040127A mov     ecx, [ebp+lpBuffer]
.text:0040127D add     ecx, [eax+3Ch]
.text:00401280 mov     edx, [ebp+var_70]
.text:00401283 imul    edx, 28h ; '('
.text:00401286 lea     eax, [ecx+edx+0F8h]
.text:0040128D mov     [ebp+var_74], eax
.text:00401290 push    0 ; lpNumberOfBytesWritten
.text:00401292 mov     ecx, [ebp+var_74]
.text:00401295 mov     edx, [ecx+10h]
.text:00401298 push    edx ; nSize
.text:00401299 mov     eax, [ebp+var_74]
.text:0040129C mov     ecx, [ebp+lpBuffer]
.text:0040129F add     ecx, [eax+14h]
.text:004012A2 push    ecx ; lpBuffer
.text:004012A3 mov     edx, [ebp+var_74]
.text:004012A6 mov     eax, [ebp+lpBaseAddress]
.text:004012A9 add     eax, [edx+0Ch]
.text:004012AC push    eax ; lpBaseAddress
.text:004012AD mov     ecx, [ebp+ProcessInformation.hProcess]
.text:004012B0 push    ecx ; hProcess
.text:004012B1 call    ds:WriteProcessMemory
.text:004012B7 jmp     short loc_401260

.text:004012B9
.text:004012B9 loc_4012B9: ; lpNumberOfBytesWritten
.text:004012B9 push    0
.text:004012BB push    4 ; nSize
.text:004012BD mov     edx, [ebp+var_8]
.text:004012C0 add     edx, 34h ; '4'
.text:004012C3 push    edx ; lpBuffer
.text:004012C4 mov     eax, [ebp+lpContext]
.text:004012C7 mov     ecx, [eax+0A4h]
.text:004012CD add     ecx, 8
.text:004012D0 push    ecx ; lpBaseAddress
.text:004012D1 mov     edx, [ebp+ProcessInformation.hProcess]
.text:004012D4 push    edx ; hProcess
.text:004012D5 call    ds:WriteProcessMemory
.text:004012DB mov     eax, [ebp+var_8]
.text:004012DE mov     ecx, [ebp+lpBaseAddress]
.text:004012E1 add     ecx, [eax+28h]
.text:004012E4 mov     edx, [ebp+lpContext]
.text:004012E7 mov     [edx+0B0h], ecx
.text:004012ED mov     eax, [ebp+lpContext]
.text:004012F0 push    eax ; lpContext
.text:004012F1 mov     ecx, [ebp+ProcessInformation.hThread]
.text:004012F4 push    ecx ; hThread
.text:004012F5 call    ds:SetThreadContext
.text:004012FB mov     edx, [ebp+ProcessInformation.hThread]
.text:004012FE push    edx ; hThread
.text:004012FF call    ds:ResumeThread
.text:00401305 jmp     short loc_40130B
```

In order to fully analyze the malware, we have to find the data that gets resolved and hollowed into svchost.exe.

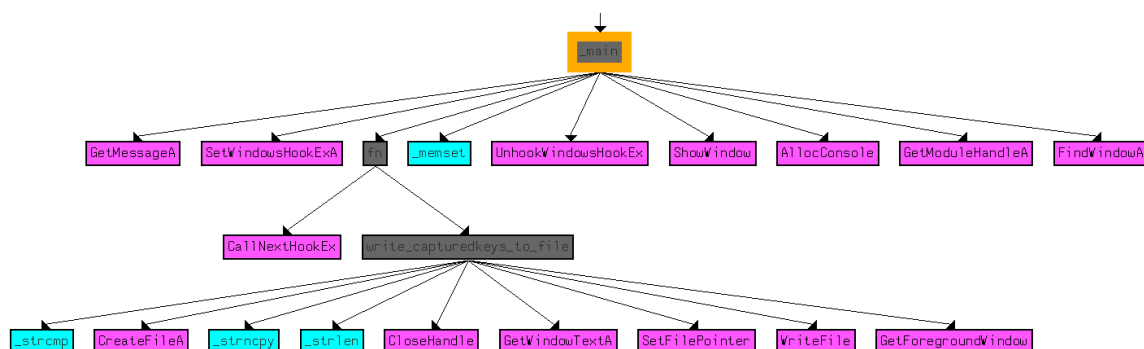
To get it, we have to locate the data after its being resolved and decoded and then dump it.

# Dump PE file

The dumped PE file seems not protected by any additional layer of obfuscation or packing.

When it comes to its behavior, it captures keystrokes through hooking using `SetWindowsHookExA` with an id `0x0D` that stands for `WH_KEYBOARD_LL` which confirms our previous dynamic analysis when we saw each pressed key being saved into the file.

The **`SetWindowsHookExA`** performs a hook procedure that IDA named as **`fn`**, that records all of the keys and its correlated window names and after all it saves the output continuously to a file “`practicalmalwareanalysis.log`”



# Questions

## 1. What is the purpose of this program?

The purpose of this malware is to record all of the keystrokes and its corresponding window names. It's a loader and keylogger type of malware.

## 2. How does the launcher program hide execution?

The launcher hides execution by performing process hollowing.

It creates a suspended svchost.exe process, unmaps existing memory and overwrites it with a completely different code of its own PE file and resumes the process.

## 3. Where is the malicious payload stored?

The malicious payload is hidden in resources section under **UNICODE/LOCALIZATION** of the provided in labs PE file.

At the stage of execution it gets resolved from the section and then mapped into the victim process.

## 4. How is the malicious payload protected?

The malicious encoded payload is first resolved from the resources at 0x40132C and then randomly allocated in the memory.

The decoding function takes place at 0x401000, each of the byte is XOR-encoded by 0x41, so the code just performs XOR operation on each byte of the resolved resource with 0x41.

Address	Hex	ASCII
008C0000	0C 1B D1 41 42 41 41 41 45 41 41 41 BE BE 41 41	..NABAAAAAA%AA
008C0010	F9 41 41 41 41 41 41 41 01 41 41 41 41 41 41 41	UAAAAAAAA.AAAAAA
008C0020	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
008C0030	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAA;AA
008C0040	4F 5E FB 4F 41 F5 48 8C 60 F9 40 0D 8C 60 15 29	O'00A0H.'u@...)
008C0050	28 32 61 31 33 2E 26 33 20 2C 61 22 20 2F 2F 2E	(2a13.&3,'a'//.
008C0060	35 61 23 24 61 33 34 2F 61 28 2F 61 05 0E 12 61	5a#\$a34/a(/a...a
008C0070	2C 2E 25 24 6F 4C 4C 4B 65 41 41 41 41 41 41 41	..%\$oLLKeAAAAAA
008C0080	56 8C 8A D0 12 ED E4 83 12 ED E4 83 12 ED E4 83	V...D.1a..1a..1a.
008C0090	FA F2 EF 83 13 ED E4 83 FA F2 EE 83 01 ED E4 83	u0i...1a.u0i...1a.
008C00A0	91 F1 EA 83 18 ED E4 83 70 F2 F7 83 17 ED E4 83	.ñe...1a.p0-...1a.
008C00B0	12 ED E5 83 23 ED E4 83 FA F2 F1 83 13 ED E4 83	.1a.#1a.u0ñ...1a.
008C00C0	13 28 22 29 12 ED E4 83 41 41 41 41 41 41 41 41	.(").1a.AAAAAAA
008C00D0	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
008C00E0	11 04 41 41 0D 40 42 41 22 10 C1 0C 41 41 41 41	..AA.@BA".A.AAAA
008C00F0	41 41 41 41 A1 41 4E 40 4A 40 47 41 41 71 41 41	AAAA;AN@J@GAAQAA
008C0100	41 71 41 41 41 41 41 41 15 56 41 41 41 51 41 41	AqAAAAAA.VAAQAA
008C0110	41 01 41 41 41 41 01 41 41 51 41 41 41 51 41 41	A.AAAA.AAQAAQAA
008C0120	45 41 41 41 41 41 41 41 45 41 41 41 41 41 41 41	EAAAAAAAAAAAAAAAAA
008C0130	41 31 41 41 41 41 51 41 41 41 41 41 41 42 41 41	A1AAQAQAAAAABAAA
008C0140	41 41 51 41 41 51 41 41 41 41 51 41 41 41 41 41	QAAAAQAAQAA
008C0150	41 41 41 41 51 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
008C0160	6D 05 41 41 7D 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
008C0170	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
008C0180	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
<b>BEFORE</b>		
Address	Hex	ASCII
008C0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
008C0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	.....@.....
008C0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
008C0030	00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00	.....à.....
008C0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...".!Li!Th
008C0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program cannot
008C0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
008C0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$......
008C0080	17 CD CB 91 53 AC A5 C2 53 AC A5 C2 53 AC A5 C2	.IE.S-¥As-¥As-¥A
008C0090	BB B3 AE C2 52 AC A5 C2 BB B3 AF C2 40 AC A5 C2	»"®AR-¥A»"®A@-¥A
008C00A0	D0 B0 AB C2 59 AC A5 C2 31 B3 B6 C2 56 AC A5 C2	D®«AY-¥A1"®AV-¥A
008C00B0	53 AC A4 C2 62 AC A5 C2 BB B3 B0 C2 52 AC A5 C2	S-®Ab-¥A»"®AR-¥A
008C00C0	52 69 63 68 53 AC A5 C2 00 00 00 00 00 00 00 00	Richs-¥A.....
008C00D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
008C00E0	50 45 00 00 4C 01 03 00 63 51 80 4D 00 00 00 00	PE..L...cQ.M....
008C00F0	00 00 00 00 E0 00 0F 01 0B 01 06 00 00 30 00 00	...à.....0...
008C0100	00 30 00 00 00 00 00 00 54 17 00 00 00 10 00 00	.0.....T.....
008C0110	00 40 00 00 00 00 40 00 00 10 00 00 00 10 00 00	@.....@.....
008C0120	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	.....
008C0130	00 70 00 00 00 10 00 00 00 00 00 00 03 00 00 00	p.....
008C0140	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	.....
008C0150	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00	.....
008C0160	2C 44 00 00 3C 00 00 00 00 00 00 00 00 00 00 00	.....
008C0170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
008C0180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
<b>AFTER</b>		

### 5. How are strings protected?

As explained above – the strings are XOR-encoded with value of 0x41, to extract them we have to perform the same operation.